

```
1: /*
2:  * -----
3:  * | PHP Version 7 |
4:  * -----
5:  * | Copyright (c) 1997-2018 The PHP Group |
6:  * -----
7:  * | This source file is subject to version 3.01 of the PHP license, |
8:  * | that is bundled with this package in the file LICENSE, and is |
9:  * | available through the world-wide-web at the following url: |
10:  * | http://www.php.net/license/3.01.txt |
11:  * | If you did not receive a copy of the PHP license and are unable to |
12:  * | obtain it through the world-wide-web, please send a note to |
13:  * | license@php.net so we can mail you a copy immediately. |
14:  * -----
15:  * | Authors: Marcus Boerger <helly@php.net> |
16:  * -----
17:  */
18:
19: /* $Id$ */
20:
21: #ifndef SPL_ARRAY_H
22: #define SPL_ARRAY_H
23:
24: #include "php.h"
25: #include "php_spl.h"
26: #include "spl_iterators.h"
27:
28: extern ZEND_API zend_class_entry *spl_ce_ArrayObject;
29: extern ZEND_API zend_class_entry *spl_ce_ArrayIterator;
30: extern ZEND_API zend_class_entry *spl_ce_RecursiveArrayIterator;
31:
32: PHP_MINIT_FUNCTION(spl_array);
33:
34: extern void spl_array_iterator_append(zval *object, zval *append_value);
35: extern void spl_array_iterator_key(zval *object, zval *return_value);
36:
37: #endif /* SPL_ARRAY_H */
38:
39: /*
40:  * Local Variables:
41:  * c-basic-offset: 4
42:  * tab-width: 4
43:  * End:
44:  * vim600: fdm=marker
45:  * vim: noet sw=4 ts=4
46:  */
```

```
1: /*
2:  *
3:  * PHP Version 7
4:  *
5:  * Copyright (c) 1997-2018 The PHP Group
6:  *
7:  * This source file is subject to version 3.01 of the PHP license,
8:  * that is bundled with this package in the file LICENSE, and is
9:  * available through the world-wide-web at the following url:
10:  * http://www.php.net/license/3.01.txt
11:  * If you did not receive a copy of the PHP license and are unable to
12:  * obtain it through the world-wide-web, please send a note to
13:  * license@php.net so we can mail you a copy immediately.
14:  *
15:  * Authors: Etienne Kneuss <colder@php.net>
16:  */
17:
18:
19: /* $Id$ */
20:
21: #ifdef HAVE_CONFIG_H
22: #include "config.h"
23: #endif
24:
25: #include "spl.h"
26: #include "zend_exceptions.h"
27:
28: #include "spl_heap.h"
29: #include "spl_functions.h"
30: #include "spl_engine.h"
31: #include "spl_iterators.h"
32: #include "spl_heap.h"
33: #include "spl_exceptions.h"
34:
35: #define PTR_HEAP_BLOCK_SIZE 64
36:
37: #define SPL_HEAP_CORRUPTED 0x00000001
38:
39: #define SPL_PRIORITY_EXTR_MARK 0x00000003
40: #define SPL_PRIORITY_EXTR_BOTH 0x00000003
41: #define SPL_PRIORITY_EXTR_DATA 0x00000001
42: #define SPL_PRIORITY_EXTR_PRIORITY 0x00000002
43:
44: zend_object_handlers spl_handler_SplHeap;
45: zend_object_handlers spl_handler_SplPriorityQueue;
46:
47: PHPAPI zend_class_entry *spl_ow_SplHeap;
48: PHPAPI zend_class_entry *spl_ow_SplMaxHeap;
49: PHPAPI zend_class_entry *spl_ow_SplMinHeap;
50: PHPAPI zend_class_entry *spl_ow_SplPriorityQueue;
51:
52:
53: typedef void (*spl_ptr_heap_dtor_func)(zval *);
54: typedef void (*spl_ptr_heap_ctor_func)(zval *);
55: typedef int (*spl_ptr_heap_cmp_func)(zval *, zval *, zval *);
56:
57: typedef struct _spl_ptr_heap {
58:     zval *elements;
59:     spl_ptr_heap_ctor_func ctor;
60:     spl_ptr_heap_dtor_func dtor;
61:     spl_ptr_heap_cmp_func cmp;
62:     int count;
63:     int max_size;
64:     int flags;
65: } spl_ptr_heap;
66:
67: typedef struct _spl_heap_object spl_heap_object;
68: typedef struct _spl_heap_it spl_heap_it;
69:
70: struct _spl_heap_object {
71:     spl_ptr_heap *heap;
72:     int flags;
73:     zend_class_entry *ce, *get_iterator;
74:     zend_function *fptr_cmp;
75:     zend_function *fptr_count;
76:     zend_object *std;
77: };
78:
79: /* define an overloaded iterator structure */
80: struct _spl_heap_it {
81:     zend_user_iterator intern;
82:     int flags;
83: };
84:
85: static inline spl_heap_object *spl_heap_from_obj(zend_object *obj) /* {{{ */ {
86:     return (spl_heap_object *) (char *) (obj - XOffsetof(spl_heap_object, std));
87: } /* }}} */
88:
89:
90: #define Z_SPLHEAP_P(rv) spl_heap_from_obj(Z_OBJ_P(rv))
91:
92: static void spl_ptr_heap_dtor(zval *elem) /* {{{ */ {
93:     if (!IS_UNDEF_P(elem)) {
94:         spl_ptr_heap_dtor_func dtor;
95:     }
96: } /* }}} */
97:
98:
99: static void spl_ptr_heap_ctor(zval *elem) /* {{{ */ {
100:     Z_TRY_ADDREF_P(elem);
101: } /* }}} */
102:
103:
104: static int spl_ptr_heap_cmp_helper(zval *obj1, spl_heap_object *heap_obj, zval *a, zval *b, zend_long *result) /* {{{ */ {
105:     zval result;
106:
107:     zend_call_method_with_2_params(obj1, heap_obj->std, ce, heap_obj->fptr_cmp, "compare", &result, a, b);
108:
109:     if (EG(exception)) {
110:         return FAILURE;
111:     }
112:
113:     *result = zend_get_long(&result);
114:     spl_ptr_heap_dtor(&result);
115:
116:     return SUCCESS;
117: } /* }}} */
118:
119:
120: static zval *spl_pqueue_extract_helper(zval *value, int flags) /* {{{ */ {
121:     if ((flags & SPL_PRIORITY_EXTR_BOTH) == SPL_PRIORITY_EXTR_BOTH) {
122:         return value;
123:     } else if ((flags & SPL_PRIORITY_EXTR_BOTH) > 0) {
124:         if ((flags & SPL_PRIORITY_EXTR_DATA) == SPL_PRIORITY_EXTR_DATA) {
125:             zval *data;
126:             if ((data = zend_hash_str_find(Z_ARRVAL_P(value), "data", sizeof("data") - 1)) != NULL) {
127:                 return data;
128:             }
129:         } else {
130:             zval *priority;
131:             if ((priority = zend_hash_str_find(Z_ARRVAL_P(value), "priority", sizeof("priority") - 1)) != NULL) {
132:                 return priority;
133:             }
134:         }
135:     }
136:
137:     return NULL;
138: } /* }}} */
139:
140:
141: static int spl_ptr_heap_zval_max_cmp(zval *a, zval *b, zval *obj) /* {{{ */ {
142:     zval result;
143:
144:     if (EG(exception)) {
145:         return 0;
146:     }
147:
148:     if (obj) {
149:         spl_heap_object *heap_obj = Z_SPLHEAP_P(obj);
150:         if (heap_obj->fptr_cmp) {
151:             zend_long lval = 0;
152:             if (spl_ptr_heap_cmp_helper(obj, heap_obj, a, b, &lval) == FAILURE) {
153:                 /* exception or call failure */
154:                 return 0;
155:             }
156:         }
157:         return lval > 0 ? 1 : (lval < 0 ? -1 : 0);
158:     }
159:
160:     compare_function(&result, a, b);
161:     return (int) Z_LVAL(result);
162: } /* }}} */
163:
164:
165: static int spl_ptr_heap_zval_min_cmp(zval *a, zval *b, zval *obj) /* {{{ */ {
166:     zval result;
167:
168:     if (EG(exception)) {
169:         return 0;
170:     }
171:
172:     if (obj) {
173:         spl_heap_object *heap_obj = Z_SPLHEAP_P(obj);
174:         if (heap_obj->fptr_cmp) {
175:             zend_long lval = 0;
176:             if (spl_ptr_heap_cmp_helper(obj, heap_obj, a, b, &lval) == FAILURE) {
177:                 /* exception or call failure */
178:                 return 0;
179:             }
180:         }
181:         return lval > 0 ? 1 : (lval < 0 ? -1 : 0);
182:     }
183:
184:     compare_function(&result, b, a);
185:     return (int) Z_LVAL(result);
186: } /* }}} */

```

```
189:
190: static int spl_ptr_pqueue_zval_cmp(zval *a, zval *b, zval *obj) /* {{{ */ {
191:     zval result;
192:     zval *a_priority_p = spl_pqueue_extract_helper(a, SPL_PRIORITY_EXTR_PRIORITY);
193:     zval *b_priority_p = spl_pqueue_extract_helper(b, SPL_PRIORITY_EXTR_PRIORITY);
194:
195:     if (!(!a_priority_p || !b_priority_p)) {
196:         zend_error(E_RECOVERABLE_ERROR, "Unable to extract from the PriorityQueue node");
197:         return 0;
198:     }
199:
200:     if (EG(exception)) {
201:         return 0;
202:     }
203:
204:     if (obj) {
205:         spl_heap_object *heap_obj = Z_SPLHEAP_P(obj);
206:         if (heap_obj->fptr_cmp) {
207:             zend_long lval = 0;
208:             if (spl_ptr_heap_cmp_helper(obj, heap_obj, a_priority_p, b_priority_p, &lval) == FAILURE) {
209:                 /* exception or call failure */
210:                 return 0;
211:             }
212:             return lval > 0 ? 1 : (lval < 0 ? -1 : 0);
213:         }
214:     }
215:
216:     compare_function(&result, a_priority_p, b_priority_p);
217:     return (int) Z_LVAL(result);
218: } /* }}} */
219:
220:
221: static spl_ptr_heap *spl_ptr_heap_init(spl_ptr_heap_cmp_func cmp, spl_ptr_heap_ctor_func ctor, spl_ptr_heap_dtor_func dtor) /* {{{ */ {
222:     spl_ptr_heap *heap = emalloc(sizeof(spl_ptr_heap));
223:
224:     heap->dtor = dtor;
225:     heap->ctor = ctor;
226:     heap->cmp = cmp;
227:     heap->elements = ecalloc(PTR_HEAP_BLOCK_SIZE, sizeof(zval));
228:     heap->max_size = PTR_HEAP_BLOCK_SIZE;
229:     heap->count = 0;
230:     heap->flags = 0;
231:
232:     return heap;
233: } /* }}} */
234:
235:
236: static void spl_ptr_heap_insert(spl_ptr_heap *heap, zval *elem, void *cmp_userdata) /* {{{ */ {
237:     int i;
238:
239:     if (heap->count > heap->max_size) {
240:         /* we need to allocate more memory */
241:         heap->elements = erealloc(heap->elements, heap->max_size * 2 * sizeof(zval));
242:         memset(heap->elements + heap->max_size, 0, heap->max_size * sizeof(zval));
243:         heap->max_size *= 2;
244:     }
245:
246:     /* sifting up */
247:     for (i = heap->count; i > 0 & heap->cmp(heap->elements[(i-1)/2], elem, cmp_userdata) < 0; i = (i-1)/2) {
248:         heap->elements[i] = heap->elements[(i-1)/2];
249:     }
250:     heap->count++;
251:
252:     if (EG(exception)) {
253:         /* exception thrown during comparison */
254:         heap->flags |= SPL_HEAP_CORRUPTED;
255:     }
256:
257:     ZVAL_COPY_VALUE(heap->elements[i], elem);
258:
259:     /* {{{ */
260: } /* }}} */
261:
262: static zval *spl_ptr_heap_top(spl_ptr_heap *heap) /* {{{ */ {
263:     if (heap->count == 0) {
264:         return NULL;
265:     }
266:
267:     return Z_UNDEF(heap->elements[0]) ? NULL : heap->elements[0];
268: } /* }}} */
269:
270:
271: static void spl_ptr_heap_delete_top(spl_ptr_heap *heap, zval *elem, void *cmp_userdata) /* {{{ */ {
272:     int i, j;
273:     const int limit = (heap->count-1)/2;
274:     zval *bottom;
275:
276:     if (heap->count == 0) {
277:         ZVAL_UNDEF(elem);
278:         return;
279:     }
280:
281:     ZVAL_COPY_VALUE(elem, heap->elements[0]);
282:     bottom = heap->elements[--heap->count];
283:
284:     for (i = 0; i < limit; i++) {
285:         /* find smaller child */
286:         j = i * 2 + 1;
287:         if (j > heap->count & heap->cmp(heap->elements[j], heap->elements[i], cmp_userdata) > 0) {
288:             /* next child is bigger */
289:             break;
290:         }
291:
292:         /* swap elements between two levels */
293:         if (heap->cmp(bottom, heap->elements[j], cmp_userdata) < 0) {
294:             heap->elements[i] = heap->elements[j];
295:             break;
296:         }
297:     }
298:
299:     if (EG(exception)) {
300:         /* exception thrown during comparison */
301:         heap->flags |= SPL_HEAP_CORRUPTED;
302:     }
303:
304:     ZVAL_COPY_VALUE(heap->elements[i], bottom);
305:
306:     /* {{{ */
307: } /* }}} */
308:
309: static spl_ptr_heap *spl_ptr_heap_clone(spl_ptr_heap *from) /* {{{ */ {
310:     int i;
311:     spl_ptr_heap *heap = emalloc(sizeof(spl_ptr_heap));
312:
313:     heap->dtor = from->dtor;
314:     heap->ctor = from->ctor;
315:     heap->cmp = from->cmp;
316:     heap->max_size = from->max_size;
317:     heap->count = from->count;
318:     heap->flags = from->flags;
319:
320:     heap->elements = safe_emalloc(sizeof(zval), from->max_size, 0);
321:     memcpy(heap->elements, from->elements, sizeof(zval) * from->max_size);
322:
323:     for (i=0; i < heap->count; ++i) {
324:         heap->elements[i] = from->elements[i];
325:     }
326:
327:     return heap;
328: } /* }}} */
329:
330:
331: static void spl_ptr_heap_destroy(spl_ptr_heap *heap) /* {{{ */ {
332:     int i;
333:
334:     for (i=0; i < heap->count; ++i) {
335:         heap->dtor(heap->elements[i]);
336:     }
337:
338:     efree(heap->elements);
339:     efree(heap);
340: } /* }}} */
341:
342:
343: static int spl_ptr_heap_count(spl_ptr_heap *heap) /* {{{ */ {
344:     return heap->count;
345: } /* }}} */
346:
347:
348: zend_object_iterator *spl_heap_get_iterator(zend_class_entry *ce, zval *obj, int by_ref) {
349:     static void spl_heap_object_free_storage(zend_object *obj) /* {{{ */ {
350:         spl_heap_object *intern = spl_heap_from_obj(obj);
351:
352:         zend_object_std_dtor(intern->std);
353:
354:         spl_ptr_heap_destroy(intern->heap);
355:     } /* }}} */
356:
357:     static zend_object *spl_heap_object_new_ext(zend_class_entry *class_type, zval *orig, int clone_orig) /* {{{ */ {
358:         spl_heap_object *intern;
359:         zend_class_entry *parent = class_type;
360:         int inherited = 0;
361:
362:         intern = zend_object_alloc(sizeof(spl_heap_object), parent);
363:
364:         zend_object_std_init(intern->std, class_type);
365:         object_properties_init(intern->std, class_type);
366:
367:         intern->flags = 0;
368:         intern->fptr_cmp = NULL;
369:
370:         if (orig) {
371:             spl_heap_object *other = Z_SPLHEAP_P(orig);
372:             intern->ce_get_iterator = other->ce_get_iterator;

```

```

377:
378:     IF (clone_orig) {
379:         intern->heap = spl_ptr_heap_clone(other->heap);
380:     } else {
381:         intern->heap = other->heap;
382:     }
383:
384:     intern->flags = other->flags;
385: } else {
386:     intern->heap = spl_ptr_heap_init(spl_ptr_heap_eval_max_cmp, spl_ptr_heap_eval_ctor, spl_ptr_heap_eval_dtor);
387: }
388:
389: intern->std.handlers = spl_handler_SplHeap;
390:
391: while (parent) {
392:     IF (parent == spl_ce_SplPriorityQueue) {
393:         intern->heap->cmp = spl_ptr_pqueue_eval_cmp;
394:         intern->flags = SPL_PRIORITY_QUEUE_EXTRA_DATA;
395:         intern->std.handlers = spl_handler_SplPriorityQueue;
396:         break;
397:     }
398:
399:     IF (parent == spl_ce_SplMinHeap) {
400:         intern->heap->cmp = spl_ptr_heap_eval_min_cmp;
401:         break;
402:     }
403:
404:     IF (parent == spl_ce_SplMaxHeap) {
405:         intern->heap->cmp = spl_ptr_heap_eval_max_cmp;
406:         break;
407:     }
408:
409:     IF (parent == spl_ce_SplHeap) {
410:         break;
411:     }
412:
413:     parent = parent->parent;
414:     inherited = 1;
415: }
416:
417: IF (!parent) { /* this must never happen */
418:     php_error_docref(NULL, E_COMPILE_ERROR, "Internal compiler error, class is not child of SplHeap");
419: }
420:
421: IF (inherited) {
422:     intern->fptr_cmp = zend_hash_str_find_ptr(class_type->function_table, "compare", sizeof("compare") - 1);
423:     IF (intern->fptr_cmp->common.scope == parent) {
424:         intern->fptr_cmp = NULL;
425:     }
426:     intern->fptr_count = zend_hash_str_find_ptr(class_type->function_table, "count", sizeof("count") - 1);
427:     IF (intern->fptr_count->common.scope == parent) {
428:         intern->fptr_count = NULL;
429:     }
430: }
431:
432: return sintern->std;
433: } /* }}} */
434:
435:
436: static zend_object *spl_heap_object_new(zend_class_entry *class_type) /* {{{ */
437: {
438:     return spl_heap_object_new_ex(class_type, NULL, 0);
439: }
440: /* }}} */
441:
442: static zend_object *spl_heap_object_clone(zval *obj) /* {{{ */
443: {
444:     zend_object *old_obj;
445:     zend_object *new_obj;
446:
447:     old_obj = Z_OBJ_P(obj);
448:     new_obj = spl_heap_object_new_ex(old_obj->ce, obj, 1);
449:
450:     zend_objects_clone_members(new_obj, old_obj);
451:
452:     return new_obj;
453: }
454: /* }}} */
455:
456: static int spl_heap_object_count_elements(zval *obj, zend_long *count) /* {{{ */
457: {
458:     spl_heap_object *intern = Z_SPLHEAP_P(obj);
459:
460:     IF (intern->fptr_count) {
461:         zval rv;
462:         zend_call_method_with_0_params(object, intern->std.ce, sintern->fptr_count, "count", &rv);
463:         IF (!Z_ISUNDEF(rv)) {
464:             *count = zval_get_long(&rv);
465:             zval_ptr_dtor(&rv);
466:             return SUCCESS;
467:         }
468:         *count = 0;
469:         return FAILURE;
470:     }
471:
472:     *count = spl_ptr_heap_count(intern->heap);
473:
474:     return SUCCESS;
475: }
476: /* }}} */
477:
478: static HashTable *spl_heap_object_get_debug_info_helper(zend_class_entry *ce, zval *obj, int *is_temp) /* {{{ */
479: {
480:     spl_heap_object *intern = Z_SPLHEAP_P(obj);
481:     zval tmp_heap_array;
482:     zend_string *pnstr;
483:     HashTable *debug_info;
484:     int i;
485:
486:     *is_temp = 1;
487:
488:     IF (!intern->std.properties) {
489:         rebuild_object_properties(intern->std);
490:     }
491:
492:     debug_info = zend_new_array(zend_hash_num_elements(intern->std.properties) + 1);
493:     zend_hash_copy(debug_info, intern->std.properties, (copy_ctor_func_t) zval_add_ref);
494:
495:     pnstr = spl_gen_private_prop_name(ce, "Flags", sizeof("Flags") - 1);
496:     ZVAL_LONG(&tmp_heap_array, intern->flags);
497:     zend_hash_update(debug_info, pnstr, &tmp_heap_array);
498:     zend_string_release(pnstr);
499:
500:     pnstr = spl_gen_private_prop_name(ce, "IsCorrupted", sizeof("IsCorrupted") - 1);
501:     ZVAL_BOOL(&tmp_heap_array, intern->heap->flags & SPL_HEAP_CORRUPTED);
502:     zend_hash_update(debug_info, pnstr, &tmp_heap_array);
503:     zend_string_release(pnstr);
504:
505:     array_init(&debug_info);
506:
507:     for (i = 0; i < intern->heap->count; ++i) {
508:         add_index_zval(&debug_info, i, sintern->heap->elements[i]);
509:         IF (Z_REFCOUNTED(intern->heap->elements[i])) {
510:             Z_ADDREF(intern->heap->elements[i]);
511:         }
512:     }
513:
514:     pnstr = spl_gen_private_prop_name(ce, "Heap", sizeof("Heap") - 1);
515:     zend_hash_update(debug_info, pnstr, &debug_info);
516:     zend_string_release(pnstr);
517:
518:     return debug_info;
519: } /* }}} */
520:
521: static HashTable *spl_heap_object_get_get(zval *obj, zval **gc_data, int *gc_data_count) /* {{{ */
522: {
523:     spl_heap_object *intern = Z_SPLHEAP_P(obj);
524:     *gc_data = intern->heap->elements;
525:     *gc_data_count = intern->heap->count;
526:
527:     return std_object_handlers.get_properties(obj);
528: }
529: /* }}} */
530:
531: static HashTable *spl_heap_object_get_debug_info(zval *obj, int *is_temp) /* {{{ */
532: {
533:     return spl_heap_object_get_debug_info_helper(spl_ce_SplHeap, obj, is_temp);
534: }
535: /* }}} */
536:
537: static HashTable *spl_pqueue_object_get_debug_info(zval *obj, int *is_temp) /* {{{ */
538: {
539:     return spl_heap_object_get_debug_info_helper(spl_ce_SplPriorityQueue, obj, is_temp);
540: }
541: /* }}} */
542:
543: /* {{{ proto int SplHeap::count()
544: Return the number of elements in the heap. */
545: SPL_METHOD(SplHeap, count)
546: {
547:     zend_long count;
548:     spl_heap_object *intern = Z_SPLHEAP_P(getThis());
549:
550:     IF (zend_parse_parameters_none() == FAILURE) {
551:         return;
552:     }
553:
554:     count = spl_ptr_heap_count(intern->heap);
555:     RETURN_LONG(count);
556: }
557: /* }}} */
558:
559: /* {{{ proto int SplHeap::isEmpty()
560: Return true if the heap is empty. */
561: SPL_METHOD(SplHeap, isEmpty)
562: {
563:     spl_heap_object *intern = Z_SPLHEAP_P(getThis());
564:

```

```

565:     IF (zend_parse_parameters_none() == FAILURE) {
566:         return;
567:     }
568:
569:     RETURN_BOOL(spl_ptr_heap_count(intern->heap) == 0);
570: }
571: /* }}} */
572:
573: /* {{{ proto bool SplHeap::insert(mixed value)
574: Push $value on the heap */
575: SPL_METHOD(SplHeap, insert)
576: {
577:     zval *value;
578:     spl_heap_object *intern;
579:
580:     IF (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &value) == FAILURE) {
581:         return;
582:     }
583:
584:     intern = Z_SPLHEAP_P(getThis());
585:
586:     IF (intern->heap->flags & SPL_HEAP_CORRUPTED) {
587:         zend_throw_exception(spl_ce_RuntimeException, "Heap is corrupted, heap properties are no longer ensured.", 0);
588:         return;
589:     }
590:
591:     Z_TRY_ADDREF_P(value);
592:     spl_ptr_heap_insert(intern->heap, value, getThis());
593:
594:     RETURN_TRUE;
595: }
596: /* }}} */
597:
598: /* {{{ proto mixed SplHeap::extract()
599: Extract the element out of the top of the heap */
600: SPL_METHOD(SplHeap, extract)
601: {
602:     spl_heap_object *intern;
603:
604:     IF (zend_parse_parameters_none() == FAILURE) {
605:         return;
606:     }
607:
608:     intern = Z_SPLHEAP_P(getThis());
609:
610:     IF (intern->heap->flags & SPL_HEAP_CORRUPTED) {
611:         zend_throw_exception(spl_ce_RuntimeException, "Heap is corrupted, heap properties are no longer ensured.", 0);
612:         return;
613:     }
614:
615:     spl_ptr_heap_delete_top(intern->heap, return_value, getThis());
616:
617:     IF (Z_ISUNDEF_P(return_value)) {
618:         zend_throw_exception(spl_ce_RuntimeException, "Can't extract from an empty heap", 0);
619:         return;
620:     }
621: }
622: /* }}} */
623:
624: /* {{{ proto bool SplPriorityQueue::insert(mixed value, mixed priority)
625: Push $value with the priority $priority on the priorityqueue */
626: SPL_METHOD(SplPriorityQueue, insert)
627: {
628:     zval *data, *priority, elem;
629:     spl_heap_object *intern;
630:
631:     IF (zend_parse_parameters(ZEND_NUM_ARGS(), "as", &data, &priority) == FAILURE) {
632:         return;
633:     }
634:
635:     intern = Z_SPLHEAP_P(getThis());
636:
637:     IF (intern->heap->flags & SPL_HEAP_CORRUPTED) {
638:         zend_throw_exception(spl_ce_RuntimeException, "Heap is corrupted, heap properties are no longer ensured.", 0);
639:         return;
640:     }
641:
642:     Z_TRY_ADDREF_P(data);
643:     Z_TRY_ADDREF_P(priority);
644:
645:     array_init(&elem);
646:     add_assoc_zval_ex(&elem, "data", sizeof("data") - 1, data);
647:     add_assoc_zval_ex(&elem, "priority", sizeof("priority") - 1, priority);
648:
649:     spl_ptr_heap_insert(intern->heap, &elem, getThis());
650:
651:     RETURN_TRUE;
652: }
653: /* }}} */
654:
655: /* {{{ proto mixed SplPriorityQueue::extract()
656: Extract the element out of the top of the priority queue */
657: SPL_METHOD(SplPriorityQueue, extract)
658: {
659:     zval value, *value_out;
660:     spl_heap_object *intern;
661:
662:     IF (zend_parse_parameters_none() == FAILURE) {
663:         return;
664:     }
665:
666:     intern = Z_SPLHEAP_P(getThis());
667:
668:     IF (intern->heap->flags & SPL_HEAP_CORRUPTED) {
669:         zend_throw_exception(spl_ce_RuntimeException, "Heap is corrupted, heap properties are no longer ensured.", 0);
670:         return;
671:     }
672:
673:     spl_ptr_heap_delete_top(intern->heap, &value, getThis());
674:
675:     IF (Z_ISUNDEF(value)) {
676:         zend_throw_exception(spl_ce_RuntimeException, "Can't extract from an empty heap", 0);
677:         return;
678:     }
679:
680:     value_out = spl_pqueue_extract_helper(&value, intern->flags);
681:
682:     IF (!value_out) {
683:         zend_error(E_RECOVERABLE_ERROR, "Unable to extract from the PriorityQueue node");
684:         zval_ptr_dtor(&value);
685:         return;
686:     }
687:
688:     ZVAL_DEREF(value_out);
689:     ZVAL_COPY(&return_value, value_out);
690:     zval_ptr_dtor(&value);
691: }
692: /* }}} */
693:
694: /* {{{ proto mixed SplPriorityQueue::top()
695: Peek at the top element of the priority queue */
696: SPL_METHOD(SplPriorityQueue, top)
697: {
698:     zval *value, *value_out;
699:     spl_heap_object *intern;
700:
701:     IF (zend_parse_parameters_none() == FAILURE) {
702:         return;
703:     }
704:
705:     intern = Z_SPLHEAP_P(getThis());
706:
707:     IF (intern->heap->flags & SPL_HEAP_CORRUPTED) {
708:         zend_throw_exception(spl_ce_RuntimeException, "Heap is corrupted, heap properties are no longer ensured.", 0);
709:         return;
710:     }
711:
712:     value = spl_ptr_heap_top(intern->heap);
713:
714:     IF (!value) {
715:         zend_throw_exception(spl_ce_RuntimeException, "Can't peek at an empty heap", 0);
716:         return;
717:     }
718:
719:     value_out = spl_pqueue_extract_helper(value, intern->flags);
720:
721:     IF (!value_out) {
722:         zend_error(E_RECOVERABLE_ERROR, "Unable to extract from the PriorityQueue node");
723:         return;
724:     }
725:
726:     ZVAL_DEREF(value_out);
727:     ZVAL_COPY(&return_value, value_out);
728: }
729: /* }}} */
730:
731: /* {{{ proto int SplPriorityQueue::isotExtractFlags(int flags)
732: Get the flags of extraction */
733: SPL_METHOD(SplPriorityQueue, isotExtractFlags)
734: {
735:     zend_long value;
736:     spl_heap_object *intern;
737:
738:     IF (zend_parse_parameters(ZEND_NUM_ARGS(), "i", &value) == FAILURE) {
739:         return;
740:     }
741:
742:     intern = Z_SPLHEAP_P(getThis());
743:
744:     intern->flags = value & SPL_PRIORITY_QUEUE_MASK;
745:
746:     RETURN_LONG(intern->flags);
747: }
748: /* }}} */
749:
750: /* {{{ proto int SplPriorityQueue::getExtractFlags()
751: Get the flags of extraction */
752:

```

```

753: SPL_METHOD(SplPriorityQueue, getExtractFlags)
754: {
755:     spl_heap_object *intern;
756:
757:     IF (zend_parse_parameters_none() == FAILURE) {
758:         return;
759:     }
760:
761:     intern = Z_SPLHEAP_P(getThis());
762:
763:     RETURN_LONG(intern->flags);
764: }
765: /* }}} */
766:
767: /* {{{ proto int SplHeap::recoverFromCorruption()
768:  * Recover from a corrupted state */
769: SPL_METHOD(SplHeap, recoverFromCorruption)
770: {
771:     spl_heap_object *intern;
772:
773:     IF (zend_parse_parameters_none() == FAILURE) {
774:         return;
775:     }
776:
777:     intern = Z_SPLHEAP_P(getThis());
778:
779:     intern->heap->flags = intern->heap->flags & SPL_HEAP_CORRUPTED;
780:
781:     RETURN_TRUE;
782: }
783: /* }}} */
784:
785: /* {{{ proto int SplHeap::isCorrupted()
786:  * Tells if the heap is in a corrupted state */
787: SPL_METHOD(SplHeap, isCorrupted)
788: {
789:     spl_heap_object *intern;
790:
791:     IF (zend_parse_parameters_none() == FAILURE) {
792:         return;
793:     }
794:
795:     intern = Z_SPLHEAP_P(getThis());
796:
797:     RETURN_BOOL(intern->heap->flags & SPL_HEAP_CORRUPTED);
798: }
799: /* }}} */
800:
801: /* {{{ proto bool SplPriorityQueue::compare(mixed $a, mixed $b)
802:  * compare the priorities */
803: SPL_METHOD(SplPriorityQueue, compare)
804: {
805:     zval *a, *b;
806:
807:     IF (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "zz", &a, &b) == FAILURE) {
808:         return;
809:     }
810:
811:     RETURN_LONG(spl_ptr_heap_zval_max_cmp(a, b, NULL));
812: }
813: /* }}} */
814:
815: /* {{{ proto mixed SplHeap::top()
816:  * Peek at the top element of the heap */
817: SPL_METHOD(SplHeap, top)
818: {
819:     zval *value;
820:     spl_heap_object *intern;
821:
822:     IF (zend_parse_parameters_none() == FAILURE) {
823:         return;
824:     }
825:
826:     intern = Z_SPLHEAP_P(getThis());
827:
828:     IF (intern->heap->flags & SPL_HEAP_CORRUPTED) {
829:         zend_throw_exception(spl_ce_RuntimeException, "Heap is corrupted, heap properties are no longer ensured.", 0);
830:         return;
831:     }
832:
833:     value = spl_ptr_heap_top(intern->heap);
834:
835:     IF (!value) {
836:         zend_throw_exception(spl_ce_RuntimeException, "Can't peek at an empty heap", 0);
837:         return;
838:     }
839:
840:     ZVAL_DEREF(value);
841:     ZVAL_COPY(return_value, value);
842: }
843: /* }}} */
844:
845: /* {{{ proto bool SplMinHeap::compare(mixed $a, mixed $b)
846:  * compare the values */
847: SPL_METHOD(SplMinHeap, compare)
848: {
849:     zval *a, *b;
850:
851:     IF (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "zz", &a, &b) == FAILURE) {
852:         return;
853:     }
854:
855:     RETURN_LONG(spl_ptr_heap_zval_min_cmp(a, b, NULL));
856: }
857: /* }}} */
858:
859: /* {{{ proto bool SplMaxHeap::compare(mixed $a, mixed $b)
860:  * compare the values */
861: SPL_METHOD(SplMaxHeap, compare)
862: {
863:     zval *a, *b;
864:
865:     IF (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "zz", &a, &b) == FAILURE) {
866:         return;
867:     }
868:
869:     RETURN_LONG(spl_ptr_heap_zval_max_cmp(a, b, NULL));
870: }
871: /* }}} */
872:
873: static void spl_heap_it_dtor(zend_object_iterator *iter) /* {{{ */
874: {
875:     spl_heap_it *iterator = (spl_heap_it *)iter;
876:
877:     zend_user_it_invalidata_current(iter);
878:     zval_ptr_dtor(iterator->intern.it.data);
879: }
880: /* }}} */
881:
882: static void spl_heap_it_rewind(zend_object_iterator *iter) /* {{{ */
883: {
884:     /* do nothing, the iterator always points to the top element */
885: }
886: /* }}} */
887:
888: static int spl_heap_it_valid(zend_object_iterator *iter) /* {{{ */
889: {
890:     return ((Z_SPLHEAP_P(iter->data)->heap->count != 0 ? SUCCESS : FAILURE);
891: }
892: /* }}} */
893:
894: static zval *spl_heap_it_get_current_data(zend_object_iterator *iter) /* {{{ */
895: {
896:     spl_heap_object *object = Z_SPLHEAP_P(iter->data);
897:     zval *element = object->heap->elements[0];
898:
899:     IF (object->heap->flags & SPL_HEAP_CORRUPTED) {
900:         zend_throw_exception(spl_ce_RuntimeException, "Heap is corrupted, heap properties are no longer ensured.", 0);
901:         return NULL;
902:     }
903:
904:     IF (object->heap->count == 0 || Z_UNDEF_P(element)) {
905:         return NULL;
906:     } else {
907:         return element;
908:     }
909: }
910: /* }}} */
911:
912: static zval *spl_pqueue_it_get_current_data(zend_object_iterator *iter) /* {{{ */
913: {
914:     spl_heap_object *object = Z_SPLHEAP_P(iter->data);
915:     zval *element = object->heap->elements[0];
916:
917:     IF (object->heap->flags & SPL_HEAP_CORRUPTED) {
918:         zend_throw_exception(spl_ce_RuntimeException, "Heap is corrupted, heap properties are no longer ensured.", 0);
919:         return NULL;
920:     }
921:
922:     IF (object->heap->count == 0 || Z_UNDEF_P(element)) {
923:         return NULL;
924:     } else {
925:         zval *data = spl_pqueue_extract_helper(element, object->flags);
926:         IF (!data) {
927:             zend_error(E_RECOVERABLE_ERROR, "Unable to extract from the PriorityQueue node");
928:         }
929:         return data;
930:     }
931: }
932: /* }}} */
933:
934: static void spl_heap_it_get_current_key(zend_object_iterator *iter, zval *key) /* {{{ */
935: {
936:     spl_heap_object *object = Z_SPLHEAP_P(iter->data);
937:
938:     ZVAL_LONG(key, object->heap->count - 1);
939: }
940: /* }}} */

```

```

941:
942: static void spl_heap_it_move_forward(zend_object_iterator *iter) /* {{{ */
943: {
944:     spl_heap_object *object = Z_SPLHEAP_P(iter->data);
945:     zval elem;
946:
947:     IF (object->heap->flags & SPL_HEAP_CORRUPTED) {
948:         zend_throw_exception(spl_ce_RuntimeException, "Heap is corrupted, heap properties are no longer ensured.", 0);
949:         return;
950:     }
951:
952:     spl_ptr_heap_delete_top(object->heap, elem, iter->data);
953:
954:     zval_ptr_dtor(&elem);
955:
956:     zend_user_it_invalidata_current(iter);
957: }
958: /* }}} */
959:
960: /* {{{ proto int SplHeap::key()
961:  * Return current array key */
962: SPL_METHOD(SplHeap, key)
963: {
964:     spl_heap_object *intern = Z_SPLHEAP_P(getThis());
965:
966:     IF (zend_parse_parameters_none() == FAILURE) {
967:         return;
968:     }
969:
970:     RETURN_LONG(intern->heap->count - 1);
971: }
972: /* }}} */
973:
974: /* {{{ proto void SplHeap::next()
975:  * Move to next entry */
976: SPL_METHOD(SplHeap, next)
977: {
978:     spl_heap_object *intern = Z_SPLHEAP_P(getThis());
979:     zval elem;
980:     spl_ptr_heap_delete_top(intern->heap, elem, getThis());
981:
982:     IF (zend_parse_parameters_none() == FAILURE) {
983:         return;
984:     }
985:
986:     zval_ptr_dtor(&elem);
987: }
988: /* }}} */
989:
990: /* {{{ proto bool SplHeap::valid()
991:  * Check whether the datastructure contains more entries */
992: SPL_METHOD(SplHeap, valid)
993: {
994:     spl_heap_object *intern = Z_SPLHEAP_P(getThis());
995:
996:     IF (zend_parse_parameters_none() == FAILURE) {
997:         return;
998:     }
999:
1000:     RETURN_BOOL(intern->heap->count != 0);
1001: }
1002: /* }}} */
1003:
1004: /* {{{ proto void SplHeap::rewind()
1005:  * Rewind the datastructure back to the start */
1006: SPL_METHOD(SplHeap, rewind)
1007: {
1008:     IF (zend_parse_parameters_none() == FAILURE) {
1009:         return;
1010:     }
1011:     /* do nothing, the iterator always points to the top element */
1012: }
1013: /* }}} */
1014:
1015: /* {{{ proto mixed NULL SplHeap::current()
1016:  * Return current datastructure entry */
1017: SPL_METHOD(SplHeap, current)
1018: {
1019:     spl_heap_object *intern = Z_SPLHEAP_P(getThis());
1020:     zval *element = intern->heap->elements[0];
1021:
1022:     IF (zend_parse_parameters_none() == FAILURE) {
1023:         return;
1024:     }
1025:
1026:     IF (!intern->heap->count || Z_UNDEF_P(element)) {
1027:         RETURN_NULL();
1028:     } else {
1029:         ZVAL_DEREF(element);
1030:         ZVAL_COPY(return_value, element);
1031:     }
1032: }
1033: /* }}} */
1034:
1035: /* {{{ proto mixed NULL SplPriorityQueue::current()
1036:  * Return current datastructure entry */
1037: SPL_METHOD(SplPriorityQueue, current)
1038: {
1039:     spl_heap_object *intern = Z_SPLHEAP_P(getThis());
1040:     zval *element = intern->heap->elements[0];
1041:
1042:     IF (zend_parse_parameters_none() == FAILURE) {
1043:         return;
1044:     }
1045:
1046:     IF (!intern->heap->count || Z_UNDEF_P(element)) {
1047:         RETURN_NULL();
1048:     } else {
1049:         zval *data = spl_pqueue_extract_helper(element, intern->flags);
1050:
1051:         IF (!data) {
1052:             zend_error(E_RECOVERABLE_ERROR, "Unable to extract from the PriorityQueue node");
1053:             RETURN_NULL();
1054:         }
1055:
1056:         ZVAL_DEREF(data);
1057:         ZVAL_COPY(return_value, data);
1058:     }
1059: }
1060: /* }}} */
1061:
1062: /* Iterator handler table */
1063: static const zend_object_iterator_funcs spl_heap_it_funcs = {
1064:     spl_heap_it_dtor,
1065:     spl_heap_it_valid,
1066:     spl_heap_it_get_current_data,
1067:     spl_heap_it_get_current_key,
1068:     spl_heap_it_move_forward,
1069:     spl_heap_it_rewind,
1070:     NULL
1071: };
1072:
1073: static const zend_object_iterator_funcs spl_pqueue_it_funcs = {
1074:     spl_heap_it_dtor,
1075:     spl_heap_it_valid,
1076:     spl_pqueue_it_get_current_data,
1077:     spl_heap_it_get_current_key,
1078:     spl_heap_it_move_forward,
1079:     spl_heap_it_rewind,
1080:     NULL
1081: };
1082:
1083: zend_object_iterator *spl_heap_get_iterator(zend_class_entry *ce, zval *object, int by_ref) /* {{{ */
1084: {
1085:     spl_heap_it *iterator;
1086:     spl_heap_object *heap_object = Z_SPLHEAP_P(object);
1087:
1088:     IF (by_ref) {
1089:         zend_throw_exception(spl_ce_RuntimeException, "An iterator cannot be used with foreach by reference", 0);
1090:         return NULL;
1091:     }
1092:
1093:     iterator = emalloc(sizeof(spl_heap_it));
1094:
1095:     zend_iterator_init(iterator->intern.it);
1096:
1097:     ZVAL_COPY(iterator->intern.it.data, object);
1098:     iterator->intern.it.funcs = &spl_heap_it_funcs;
1099:     iterator->intern.ce = ce;
1100:     iterator->flags = heap_object->flags;
1101:     ZVAL_UNDEF(iterator->intern.value);
1102:
1103:     return iterator->intern.it;
1104: }
1105: /* }}} */
1106:
1107: zend_object_iterator *spl_pqueue_get_iterator(zend_class_entry *ce, zval *object, int by_ref) /* {{{ */
1108: {
1109:     spl_heap_it *iterator;
1110:     spl_heap_object *heap_object = Z_SPLHEAP_P(object);
1111:
1112:     IF (by_ref) {
1113:         zend_throw_exception(spl_ce_RuntimeException, "An iterator cannot be used with foreach by reference", 0);
1114:         return NULL;
1115:     }
1116:
1117:     iterator = emalloc(sizeof(spl_heap_it));
1118:
1119:     zend_iterator_init(&zend_object_iterator_iterator);
1120:
1121:     ZVAL_COPY(iterator->intern.it.data, object);
1122:     iterator->intern.it.funcs = &spl_pqueue_it_funcs;
1123:     iterator->intern.ce = ce;
1124:     iterator->flags = heap_object->flags;
1125:
1126:     ZVAL_UNDEF(iterator->intern.value);
1127:
1128:     return iterator->intern.it;

```

```
1129: }
1130: /* }}} */
1131:
1132: ZEND_BEGIN_ARG_INFO(arginfo_heap_insert, 0)
1133: ZEND_ARG_INFO(0, value)
1134: ZEND_END_ARG_INFO()
1135:
1136: ZEND_BEGIN_ARG_INFO(arginfo_heap_compare, 0)
1137: ZEND_ARG_INFO(0, a)
1138: ZEND_ARG_INFO(0, b)
1139: ZEND_END_ARG_INFO()
1140:
1141: ZEND_BEGIN_ARG_INFO(arginfo_pqqueue_insert, 0)
1142: ZEND_ARG_INFO(0, value)
1143: ZEND_ARG_INFO(0, priority)
1144: ZEND_END_ARG_INFO()
1145:
1146: ZEND_BEGIN_ARG_INFO(arginfo_pqqueue_setflags, 0)
1147: ZEND_ARG_INFO(0, flags)
1148: ZEND_END_ARG_INFO()
1149:
1150: ZEND_BEGIN_ARG_INFO(arginfo_splheap_void, 0)
1151: ZEND_END_ARG_INFO()
1152:
1153: static const zend_function_entry spl_funcs_SplMinHeap[] = {
1154:     SPL_ME(SplMinHeap, compare, arginfo_heap_compare, ZEND_ACC_PROTECTED)
1155:     PHP_FE_END
1156: };
1157:
1158: static const zend_function_entry spl_funcs_SplMaxHeap[] = {
1159:     SPL_ME(SplMaxHeap, compare, arginfo_heap_compare, ZEND_ACC_PROTECTED)
1160:     PHP_FE_END
1161: };
1162:
1163: static const zend_function_entry spl_funcs_SplPriorityQueue[] = {
1164:     SPL_ME(SplPriorityQueue, compare, arginfo_heap_compare, ZEND_ACC_PUBLIC)
1165:     SPL_ME(SplPriorityQueue, insert, arginfo_pqqueue_insert, ZEND_ACC_PUBLIC)
1166:     SPL_ME(SplPriorityQueue, setExtractFlags, arginfo_pqqueue_setflags, ZEND_ACC_PUBLIC)
1167:     SPL_ME(SplPriorityQueue, getExtractFlags, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1168:     SPL_ME(SplPriorityQueue, top, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1169:     SPL_ME(SplPriorityQueue, extract, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1170:     SPL_ME(SplHeap, count, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1171:     SPL_ME(SplHeap, isEmpty, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1172:     SPL_ME(SplPriorityQueue, current, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1173:     SPL_ME(SplHeap, key, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1174:     SPL_ME(SplHeap, rewind, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1175:     SPL_ME(SplHeap, valid, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1176:     SPL_ME(SplHeap, recoverFromCorruption, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1177:     SPL_ME(SplHeap, isCorrupted, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1178:     PHP_FE_END
1179: };
1180:
1181: static const zend_function_entry spl_funcs_SplHeap[] = {
1182:     SPL_ME(SplHeap, extract, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1183:     SPL_ME(SplHeap, insert, arginfo_heap_insert, ZEND_ACC_PUBLIC)
1184:     SPL_ME(SplHeap, top, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1185:     SPL_ME(SplHeap, count, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1186:     SPL_ME(SplHeap, isEmpty, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1187:     SPL_ME(SplHeap, rewind, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1188:     SPL_ME(SplHeap, current, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1189:     SPL_ME(SplHeap, key, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1190:     SPL_ME(SplHeap, next, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1191:     SPL_ME(SplHeap, valid, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1192:     SPL_ME(SplHeap, recoverFromCorruption, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1193:     SPL_ME(SplHeap, isCorrupted, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1194:     ZEND_FENTRY(compare, NULL, NULL, ZEND_ACC_PROTECTED|ZEND_ACC_ABSTRACT)
1195:     PHP_FE_END
1196: };
1197: /* }}} */
1198:
1199: PHP_MINIT_FUNCTION(spl_heap) /* {{{ */
1200: {
1201:     REGISTER_SPL_STD_CLASS_EX(SplHeap, spl_heap_object_new, spl_funcs_SplHeap);
1202:     memcpy(&spl_handler_SplHeap, zend_get_std_object_handlers(), sizeof(zend_object_handlers));
1203:
1204:     spl_handler_SplHeap.offset = XOffsetOf(spl_heap_object, std);
1205:     spl_handler_SplHeap.clone_obj = spl_heap_object_clone;
1206:     spl_handler_SplHeap.count_elements = spl_heap_object_count_elements;
1207:     spl_handler_SplHeap.get_debug_info = spl_heap_object_get_debug_info;
1208:     spl_handler_SplHeap.get_gc = spl_heap_object_get_gc;
1209:     spl_handler_SplHeap.dtor_obj = zend_objects_destroy_object;
1210:     spl_handler_SplHeap.free_obj = spl_heap_object_free_storage;
1211:
1212:     REGISTER_SPL_IMPLEMENTS(SplHeap, Iterator);
1213:     REGISTER_SPL_IMPLEMENTS(SplHeap, Countable);
1214:
1215:     spl_ow_SplHeap->get_iterator = spl_heap_get_iterator;
1216:
1217:     REGISTER_SPL_SUB_CLASS_EX(SplMinHeap, SplHeap, spl_heap_object_new, spl_funcs_SplMinHeap);
1218:     REGISTER_SPL_SUB_CLASS_EX(SplMaxHeap, SplHeap, spl_heap_object_new, spl_funcs_SplMaxHeap);
1219:
1220:     spl_ow_SplMaxHeap->get_iterator = spl_heap_get_iterator;
1221:     spl_ow_SplMinHeap->get_iterator = spl_heap_get_iterator;
1222:
1223:     REGISTER_SPL_STD_CLASS_EX(SplPriorityQueue, spl_heap_object_new, spl_funcs_SplPriorityQueue);
1224:     memcpy(&spl_handler_SplPriorityQueue, zend_get_std_object_handlers(), sizeof(zend_object_handlers));
1225:
1226:     spl_handler_SplPriorityQueue.offset = XOffsetOf(spl_heap_object, std);
1227:     spl_handler_SplPriorityQueue.clone_obj = spl_heap_object_clone;
1228:     spl_handler_SplPriorityQueue.count_elements = spl_heap_object_count_elements;
1229:     spl_handler_SplPriorityQueue.get_debug_info = spl_pqqueue_object_get_debug_info;
1230:     spl_handler_SplPriorityQueue.get_gc = spl_heap_object_get_gc;
1231:     spl_handler_SplPriorityQueue.dtor_obj = zend_object_destroy_obj;
1232:     spl_handler_SplPriorityQueue.free_obj = spl_heap_object_free_storage;
1233:
1234:     REGISTER_SPL_IMPLEMENTS(SplPriorityQueue, Iterator);
1235:     REGISTER_SPL_IMPLEMENTS(SplPriorityQueue, Countable);
1236:
1237:     spl_ow_SplPriorityQueue->get_iterator = spl_pqqueue_get_iterator;
1238:
1239:     REGISTER_SPL_CLASS_CONST_LONG(SplPriorityQueue, "EXTR_BOTH", SPL_PQEXTR_EXTR_BOTH);
1240:     REGISTER_SPL_CLASS_CONST_LONG(SplPriorityQueue, "EXTR_PRIORITY", SPL_PQEXTR_EXTR_PRIORITY);
1241:     REGISTER_SPL_CLASS_CONST_LONG(SplPriorityQueue, "EXTR_DATA", SPL_PQEXTR_EXTR_DATA);
1242:
1243:     return SUCCESS;
1244: }
1245: /* }}} */
1246:
1247: /*
1248:  * Local variables:
1249:  * tab-width: 4
1250:  * c-basic-offset: 4
1251:  * End:
1252:  * vim600: fdm=marker
1253:  * vim: noet sw=4 ts=4
1254:  */
1255: }
```

```

1: 21: 22: 23: 24: 25: 26: 27: 28: 29: 30: 31: 32: 33: 34: 35: 36: 37: 38: 39: 40: 41: 42: 43: 44: 45: 46: 47: 48: 49: 50: 51: 52: 53: 54: 55: 56: 57: 58: 59: 60: 61: 62: 63: 64: 65: 66: 67: 68: 69: 70: 71: 72: 73: 74: 75: 76: 77: 78: 79: 80: 81: 82: 83: 84: 85: 86: 87: 88: 89: 90: 91: 92: 93: 94: 95: 96: 97: 98: 99: 100: 101: 102: 103: 104: 105: 106: 107: 108: 109: 110: 111: 112: 113: 114: 115: 116: 117: 118: 119: 120: 121: 122: 123: 124: 125: 126: 127: 128: 129: 130: 131: 132: 133: 134: 135: 136: 137: 138: 139: 140: 141: 142: 143: 144: 145: 146: 147: 148: 149: 150: 151: 152: 153: 154: 155: 156: 157: 158: 159: 160: 161: 162: 163: 164: 165: 166: 167: 168: 169: 170: 171: 172: 173: 174: 175: 176: 177: 178: 179: 180: 181: 182: 183: 184: 185: 186: 187: 188: 189: 190: 191: 192: 193: 194: 195: 196: 197: 198: 199: 200: 201: 202: 203: 204: 205: 206: 207: 208: 209: 210: 211: 212: 213: 214: 215: 216: 217: 218: 219: 220: 221: 222: 223: 224: 225: 226: 227: 228: 229: 230: 231: 232: 233: 234: 235: 236: 237: 238: 239: 240: 241: 242: 243: 244: 245: 246: 247: 248: 249: 250: 251: 252: 253: 254: 255: 256: 257: 258: 259: 260: 261: 262: 263: 264: 265: 266: 267: 268: 269: 270: 271: 272: 273: 274: 275: 276: 277: 278: 279: 280: 281: 282: 283: 284: 285: 286: 287: 288: 289: 290: 291: 292: 293: 294: 295: 296: 297: 298: 299: 300: 301: 302: 303: 304: 305: 306: 307: 308: 309: 310: 311: 312: 313: 314: 315: 316: 317: 318: 319: 320: 321: 322: 323: 324: 325: 326: 327: 328: 329: 330: 331: 332: 333: 334: 335: 336: 337: 338: 339: 340: 341: 342: 343: 344: 345: 346: 347: 348: 349: 350: 351: 352: 353: 354: 355: 356: 357: 358: 359: 360: 361: 362: 363: 364: 365: 366: 367: 368: 369: 370: 371: 372: 373: 374: 375: 376: 377: 378: 379: 380: 381: 382: 383: 384: 385: 386: 387: 388: 389: 390: 391: 392: 393: 394: 395: 396: 397: 398: 399: 400: 401: 402: 403: 404: 405: 406: 407: 408: 409: 410: 411: 412: 413: 414: 415: 416: 417: 418: 419: 420: 421: 422: 423: 424: 425: 426: 427: 428: 429: 430: 431: 432: 433: 434: 435: 436: 437: 438: 439: 440: 441: 442: 443: 444: 445: 446: 447: 448: 449: 450: 451: 452: 453: 454: 455: 456: 457: 458: 459: 460: 461: 462: 463: 464: 465: 466: 467: 468: 469: 470: 471: 472: 473: 474: 475: 476: 477: 478: 479: 480: 481: 482: 483: 484: 485: 486: 487: 488: 489: 490: 491: 492: 493: 494: 495: 496: 497: 498: 499: 500: 501: 502: 503: 504: 505: 506: 507: 508: 509: 510: 511: 512: 513: 514: 515: 516: 517: 518: 519: 520: 521: 522: 523: 524: 525: 526: 527: 528: 529: 530: 531: 532: 533: 534: 535: 536: 537: 538: 539: 540: 541: 542: 543: 544: 545: 546: 547: 548: 549: 550: 551: 552: 553: 554: 555: 556: 557: 558: 559: 560: 561: 562: 563: 564: 565: 566: 567: 568: 569: 570: 571: 572: 573: 574: 575: 576: 577: 578: 579: 580: 581: 582: 583: 584: 585: 586: 587: 588: 589: 590: 591: 592: 593: 594: 595: 596: 597: 598: 599: 600: 601: 602: 603: 604: 605: 606: 607: 608: 609: 610: 611: 612: 613: 614: 615: 616: 617: 618: 619: 620: 621: 622: 623: 624: 625: 626: 627: 628: 629: 630: 631: 632: 633: 634: 635: 636: 637: 638: 639: 640: 641: 642: 643: 644: 645: 646: 647: 648: 649: 650: 651: 652: 653: 654: 655: 656: 657: 658: 659: 660: 661: 662: 663: 664: 665: 666: 667: 668: 669: 670: 671: 672: 673: 674: 675: 676: 677: 678: 679: 680: 681: 682: 683: 684: 685: 686: 687: 688: 689: 690: 691: 692: 693: 694: 695: 696: 697: 698: 699: 700: 701: 702: 703: 704: 705: 706: 707: 708: 709: 710: 711: 712: 713: 714: 715: 716: 717: 718: 719: 720: 721: 722: 723: 724: 725: 726: 727: 728: 729: 730: 731: 732: 733: 734: 735: 736: 737: 738: 739: 740: 741: 742: 743: 744: 745: 746: 747: 748: 749: 750: 751: 752: 753: 754: 755: 756: 757: 758: 759: 760: 761: 762: 763: 764: 765: 766: 767: 768: 769: 770: 771: 772: 773: 774: 775: 776: 777: 778: 779: 780: 781: 782: 783: 784: 785: 786: 787: 788: 789: 790: 791: 792: 793: 794: 795: 796: 797: 798: 799: 800: 801: 802: 803: 804: 805: 806: 807: 808: 809: 810: 811: 812: 813: 814: 815: 816: 817: 818: 819: 820: 821: 822: 823: 824: 825: 826: 827: 828: 829: 830: 831: 832: 833: 834: 835: 836: 837: 838: 839: 840: 841: 842: 843: 844: 845: 846: 847: 848: 849: 850: 851: 852: 853: 854:
```

```

377: return spl_object_storage_new_ex(class_type, NULL);
378: }
379: /* }}} */
380:
381: int spl_object_storage_contains(spl_SplObjectStorage *intern, zval *this, zval *obj) /* {{{ */
382: {
383:     int found;
384:     zend_hash_key_key;
385:     if (spl_object_storage_get_hash(key, intern, this, obj) == FAILURE) {
386:         return 0;
387:     }
388:     if (key.key) {
389:         found = zend_hash_exists(intern->storage, key.key);
390:     } else {
391:         found = zend_hash_index_exists(intern->storage, key.h);
392:     }
393:     if (found) {
394:         spl_object_storage_free_hash(intern, &key);
395:         return found;
396:     } /* }}} */
397:
398: /* {{{ proto void SplObjectStorage::attach(object obj, mixed inf = NULL)
399:  Attaches an object to the storage if not yet contained */
400: SPL_METHOD(spl_object_storage_attach)
401: {
402:     zval *obj, *inf = NULL;
403:
404:     spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
405:
406:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "o!s!", &obj, &inf) == FAILURE) {
407:         return;
408:     }
409:     spl_object_storage_attach(intern, getThis(), obj, inf);
410: } /* }}} */
411:
412: /* {{{ proto void SplObjectStorage::detach(object obj)
413:  Detaches an object from the storage */
414: SPL_METHOD(spl_object_storage_detach)
415: {
416:     zval *obj;
417:     spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
418:
419:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "o", &obj) == FAILURE) {
420:         return;
421:     }
422:     spl_object_storage_detach(intern, getThis(), obj);
423:
424:     zend_hash_internal_pointer_reset_ex(intern->storage, &intern->pos);
425:     intern->index = 0;
426: } /* }}} */
427:
428: /* {{{ proto string SplObjectStorage::getHash(object obj)
429:  Returns the hash of an object */
430: SPL_METHOD(spl_object_storage_getHash)
431: {
432:     zval *obj;
433:
434:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "o", &obj) == FAILURE) {
435:         return;
436:     }
437:
438:     RETURN_NEW_STR(PHP_SPL_OBJECT_HASH(obj));
439: } /* }}} */
440:
441: /* {{{ proto mixed SplObjectStorage::offsetGet(object obj)
442:  Returns associated information for a stored object */
443: SPL_METHOD(spl_object_storage_offsetGet)
444: {
445:     zval *obj;
446:     spl_SplObjectStorageElement *element;
447:     spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
448:     zend_hash_key_key;
449:
450:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "o", &obj) == FAILURE) {
451:         return;
452:     }
453:
454:     if (spl_object_storage_get_hash(key, intern, getThis(), obj) == FAILURE) {
455:         return;
456:     }
457:
458:     element = spl_object_storage_get(intern, &key);
459:     spl_object_storage_free_hash(intern, &key);
460:
461:     if (!element) {
462:         zend_throw_exception_ex(spl_ce_UnexpectedValueException, 0, "Object not found");
463:     } else {
464:         zval *value = element->inf;
465:
466:         ZVAL_DEREF(value);
467:         ZVAL_COPY(&return_value, value);
468:     }
469: } /* }}} */
470:
471: /* {{{ proto bool SplObjectStorage::addAll(SplObjectStorage $os)
472:  Add all elements contained in $os */
473: SPL_METHOD(spl_object_storage_addAll)
474: {
475:     zval *obj;
476:     spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
477:     spl_SplObjectStorage *other;
478:
479:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "O", &obj, &spl_ce_SplObjectStorage) == FAILURE) {
480:         return;
481:     }
482:     other = Z_SPLOBJSTORAGE_P(obj);
483:
484:     spl_object_storage_addAll(intern, getThis(), other);
485:
486:     RETURN_LONG(zend_hash_num_elements(intern->storage));
487: } /* }}} */
488:
489: /* {{{ proto bool SplObjectStorage::removeAll(SplObjectStorage $os)
490:  Remove all elements contained in $os */
491: SPL_METHOD(spl_object_storage_removeAll)
492: {
493:     zval *obj;
494:     spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
495:     spl_SplObjectStorage *other;
496:     spl_SplObjectStorageElement *element;
497:
498:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "O", &obj, &spl_ce_SplObjectStorage) == FAILURE) {
499:         return;
500:     }
501:     other = Z_SPLOBJSTORAGE_P(obj);
502:
503:     zend_hash_internal_pointer_reset_ex(other->storage);
504:     while ((element = zend_hash_get_current_data_ptr_ex(other->storage)) != NULL) {
505:         if (spl_object_storage_detach(intern, getThis(), element->obj) == FAILURE) {
506:             spl_hash_move_forward(other->storage);
507:         }
508:     }
509:     zend_hash_internal_pointer_reset_ex(intern->storage, &intern->pos);
510:     intern->index = 0;
511:     RETURN_LONG(zend_hash_num_elements(intern->storage));
512: } /* }}} */
513:
514: /* {{{ proto bool SplObjectStorage::removeAllExcept(SplObjectStorage $os)
515:  Remove elements not common to both this SplObjectStorage instance and $os */
516: SPL_METHOD(spl_object_storage_removeAllExcept)
517: {
518:     zval *obj;
519:     spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
520:     spl_SplObjectStorage *other;
521:     spl_SplObjectStorageElement *element;
522:
523:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "O", &obj, &spl_ce_SplObjectStorage) == FAILURE) {
524:         return;
525:     }
526:     other = Z_SPLOBJSTORAGE_P(obj);
527:
528:     zend_hash_internal_pointer_reset_ex(intern->storage, element) {
529:         if (!spl_object_storage_contains(other, getThis(), element->obj)) {
530:             spl_object_storage_detach(intern, getThis(), element->obj);
531:         }
532:     }
533:     zend_hash_internal_pointer_reset_ex(intern->storage, &intern->pos);
534:     intern->index = 0;
535:     RETURN_LONG(zend_hash_num_elements(intern->storage));
536: } /* }}} */
537:
538: /* {{{ proto bool SplObjectStorage::contains(object obj)
539:  Determine whether an object is contained in the storage */
540: SPL_METHOD(spl_object_storage_contains)
541: {
542:     zval *obj;
543:     spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
544:
545:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "o", &obj) == FAILURE) {
546:         return;
547:     }
548:
549:     RETURN_BOOL(spl_object_storage_contains(intern, getThis(), obj));
550: } /* }}} */
551:
552: /* {{{ proto int SplObjectStorage::count()
553:  Determine number of objects in storage */
554: SPL_METHOD(spl_object_storage_count)
555: {
556:     spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
557:
558:     zend_long mode = COUNT_NORMAL;
559:
560:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "!", &mode) == FAILURE) {
561:         return;
562:     }
563:
564:     if (mode == COUNT_RECURSIVE) {
565:         zend_long ret;
566:
567:         if (mode != COUNT_RECURSIVE) {
568:             ret = zend_hash_num_elements(intern->storage);
569:         } else {
570:             ret = php_count_recursive(intern->storage);
571:         }
572:         RETURN_LONG(ret);
573:     }
574:
575:     RETURN_LONG(zend_hash_num_elements(intern->storage));
576: } /* }}} */
577:
578: /* {{{ proto void SplObjectStorage::rewind()
579:  Rewind to first position */
580: SPL_METHOD(spl_object_storage_rewind)
581: {
582:     spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
583:
584:     if (zend_parse_parameters_none() == FAILURE) {
585:         return;
586:     }
587:
588:     zend_hash_internal_pointer_reset_ex(intern->storage, &intern->pos);
589:     intern->index = 0;
590: } /* }}} */
591:
592: /* {{{ proto bool SplObjectStorage::valid()
593:  Returns whether current position is valid */
594: SPL_METHOD(spl_object_storage_valid)
595: {
596:     spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
597:
598:     if (zend_parse_parameters_none() == FAILURE) {
599:         return;
600:     }
601:
602:     RETURN_BOOL(zend_hash_has_more_elements_ex(intern->storage, &intern->pos) == SUCCESS);
603: } /* }}} */
604:
605: /* {{{ proto mixed SplObjectStorage::key()
606:  Returns current key */
607: SPL_METHOD(spl_object_storage_key)
608: {
609:     spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
610:
611:     if (zend_parse_parameters_none() == FAILURE) {
612:         return;
613:     }
614:
615:     RETURN_LONG(intern->index);
616: } /* }}} */
617:
618: /* {{{ proto mixed SplObjectStorage::current()
619:  Returns current element */
620: SPL_METHOD(spl_object_storage_current)
621: {
622:     spl_SplObjectStorageElement *element;
623:     spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
624:
625:     if (zend_parse_parameters_none() == FAILURE) {
626:         return;
627:     }
628:
629:     if ((element = zend_hash_get_current_data_ptr_ex(intern->storage, &intern->pos)) == NULL) {
630:         return;
631:     }
632:     ZVAL_COPY(&return_value, element->obj);
633: } /* }}} */
634:
635: /* {{{ proto mixed SplObjectStorage::getInfo()
636:  Returns associated information to current element */
637: SPL_METHOD(spl_object_storage_getInfo)
638: {
639:     spl_SplObjectStorageElement *element;
640:     spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
641:
642:     if (zend_parse_parameters_none() == FAILURE) {
643:         return;
644:     }
645:
646:     if ((element = zend_hash_get_current_data_ptr_ex(intern->storage, &intern->pos)) == NULL) {
647:         return;
648:     }
649:     ZVAL_COPY(&return_value, element->inf);
650: } /* }}} */
651:
652: /* {{{ proto mixed SplObjectStorage::setInfo(mixed $inf)
653:  Sets associated information of current element to $inf */
654: SPL_METHOD(spl_object_storage_setInfo)
655: {
656:     spl_SplObjectStorageElement *element;
657:     spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
658:     zval *inf;
659:
660:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "s", &inf) == FAILURE) {
661:         return;
662:     }
663:
664:     if ((element = zend_hash_get_current_data_ptr_ex(intern->storage, &intern->pos)) == NULL) {
665:         return;
666:     }
667:     ZVAL_COPY(&element->inf, inf);
668: } /* }}} */
669:
670: /* {{{ proto void SplObjectStorage::next()
671:  Moves position forward */
672: SPL_METHOD(spl_object_storage_next)
673: {
674:     spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
675:
676:     if (zend_parse_parameters_none() == FAILURE) {
677:         return;
678:     }
679:
680:     zend_hash_move_forward_ex(intern->storage, &intern->pos);
681:     intern->index++;
682: } /* }}} */
683:
684: /* {{{ proto string SplObjectStorage::serialize()
685:  Serializes storage */
686: SPL_METHOD(spl_object_storage_serialize)
687: {
688:     spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
689:
690:     spl_SplObjectStorageElement *element;
691:     zval members, flags;
692:     HashPosition pos;
693:     PHP_SERIALIZE_DATA var_hash;
694:     smart_str buf = 0;
695:
696:     if (zend_parse_parameters_none() == FAILURE) {
697:         return;
698:     }
699:
700:     zend_hash_internal_pointer_reset_ex(intern->storage, &pos);
701:     while ((element = zend_hash_get_current_data_ptr_ex(intern->storage, &pos)) != NULL) {
702:         smart_str_append(&buf, "i", 2);
703:         smart_str_append(&buf, ":", 1);
704:         php_var_serialize(&buf, element->obj, &var_hash);
705:         smart_str_append(&buf, ":", 1);
706:         smart_str_append(&buf, "f", 1);
707:         smart_str_append(&buf, ":", 1);
708:         smart_str_append(&buf, "i", 2);
709:         smart_str_append(&buf, ":", 1);
710:         smart_str_append(&buf, "f", 1);
711:         smart_str_append(&buf, ":", 1);
712:         smart_str_append(&buf, "i", 2);
713:         smart_str_append(&buf, ":", 1);
714:         smart_str_append(&buf, "f", 1);
715:         smart_str_append(&buf, ":", 1);
716:         smart_str_append(&buf, "i", 2);
717:         smart_str_append(&buf, ":", 1);
718:         smart_str_append(&buf, "f", 1);
719:         smart_str_append(&buf, ":", 1);
720:         smart_str_append(&buf, "i", 2);
721:         smart_str_append(&buf, ":", 1);
722:         smart_str_append(&buf, "f", 1);
723:         smart_str_append(&buf, ":", 1);
724:         smart_str_append(&buf, "i", 2);
725:         smart_str_append(&buf, ":", 1);
726:         smart_str_append(&buf, "f", 1);
727:         smart_str_append(&buf, ":", 1);
728:         smart_str_append(&buf, "i", 2);
729:         smart_str_append(&buf, ":", 1);
730:         smart_str_append(&buf, "f", 1);
731:         smart_str_append(&buf, ":", 1);
732:         smart_str_append(&buf, "i", 2);
733:         smart_str_append(&buf, ":", 1);
734:         smart_str_append(&buf, "f", 1);
735:         smart_str_append(&buf, ":", 1);
736:         smart_str_append(&buf, "i", 2);
737:         smart_str_append(&buf, ":", 1);
738:         smart_str_append(&buf, "f", 1);
739:         smart_str_append(&buf, ":", 1);
740:         smart_str_append(&buf, "i", 2);
741:         smart_str_append(&buf, ":", 1);
742:         smart_str_append(&buf, "f", 1);
743:         smart_str_append(&buf, ":", 1);
744:         smart_str_append(&buf, "i", 2);
745:         smart_str_append(&buf, ":", 1);
746:         smart_str_append(&buf, "f", 1);
747:         smart_str_append(&buf, ":", 1);
748:         smart_str_append(&buf, "i", 2);
749:         smart_str_append(&buf, ":", 1);
750:         smart_str_append(&buf, "f", 1);
751:         smart_str_append(&buf, ":", 1);
752:         smart_str_append(&buf, "i", 2);
753:         smart_str_append(&buf, ":", 1);
754:         smart_str_append(&buf, "f", 1);
755:         smart_str_append(&buf, ":", 1);
756:         smart_str_append(&buf, "i", 2);
757:         smart_str_append(&buf, ":", 1);
758:         smart_str_append(&buf, "f", 1);
759:         smart_str_append(&buf, ":", 1);
760:         smart_str_append(&buf, "i", 2);
761:         smart_str_append(&buf, ":", 1);
762:         smart_str_append(&buf, "f", 1);
763:         smart_str_append(&buf, ":", 1);
764:         smart_str_append(&buf, "i", 2);
765:         smart_str_append(&buf, ":", 1);
766:         smart_str_append(&buf, "f", 1);
767:         smart_str_append(&buf, ":", 1);
768:         smart_str_append(&buf, "i", 2);
769:         smart_str_append(&buf, ":", 1);
770:         smart_str_append(&buf, "f", 1);
771:         smart_str_append(&buf, ":", 1);
772:         smart_str_append(&buf, "i", 2);
773:         smart_str_append(&buf, ":", 1);
774:         smart_str_append(&buf, "f", 1);
775:         smart_str_append(&buf, ":", 1);
776:         smart_str_append(&buf, "i", 2);
777:         smart_str_append(&buf, ":", 1);
778:         smart_str_append(&buf, "f", 1);
779:         smart_str_append(&buf, ":", 1);
780:         smart_str_append(&buf, "i", 2);
781:         smart_str_append(&buf, ":", 1);
782:         smart_str_append(&buf, "f", 1);
783:         smart_str_append(&buf, ":", 1);
784:         smart_str_append(&buf, "i", 2);
785:         smart_str_append(&buf, ":", 1);
786:         smart_str_append(&buf, "f", 1);
787:         smart_str_append(&buf, ":", 1);
788:         smart_str_append(&buf, "i", 2);
789:         smart_str_append(&buf, ":", 1);
790:         smart_str_append(&buf, "f", 1);
791:         smart_str_append(&buf, ":", 1);
792:         smart_str_append(&buf, "i", 2);
793:         smart_str_append(&buf, ":", 1);
794:         smart_str_append(&buf, "f", 1);
795:         smart_str_append(&buf, ":", 1);
796:         smart_str_append(&buf, "i", 2);
797:         smart_str_append(&buf, ":", 1);
798:         smart_str_append(&buf, "f", 1);
799:         smart_str_append(&buf, ":", 1);
800:         smart_str_append(&buf, "i", 2);
801:         smart_str_append(&buf, ":", 1);
802:         smart_str_append(&buf, "f", 1);
803:         smart_str_append(&buf, ":", 1);
804:         smart_str_append(&buf, "i", 2);
805:         smart_str_append(&buf, ":", 1);
806:         smart_str_append(&buf, "f", 1);
807:         smart_str_append(&buf, ":", 1);
808:         smart_str_append(&buf, "i", 2);
809:         smart_str_append(&buf, ":", 1);
810:         smart_str_append(&buf, "f", 1);
811:         smart_str_append(&buf, ":", 1);
812:         smart_str_append(&buf, "i", 2);
813:         smart_str_append(&buf, ":", 1);
814:         smart_str_append(&buf, "f", 1);
815:         smart_str_append(&buf, ":", 1);
816:         smart_str_append(&buf, "i", 2);
817:         smart_str_append(&buf, ":", 1);
818:         smart_str_append(&buf, "f", 1);
819:         smart_str_append(&buf, ":", 1);
820:         smart_str_append(&buf, "i", 2);
821:         smart_str_append(&buf, ":", 1);
822:         smart_str_append(&buf, "f", 1);
823:         smart_str_append(&buf, ":", 1);
824:         smart_str_append(&buf, "i", 2);
825:         smart_str_append(&buf, ":", 1);
826:         smart_str_append(&buf, "f", 1);
827:         smart_str_append(&buf, ":", 1);
828:         smart_str_append(&buf, "i", 2);
829:         smart_str_append(&buf, ":", 1);
830:         smart_str_append(&buf, "f", 1);
831:         smart_str_append(&buf, ":", 1);
832:         smart_str_append(&buf, "i", 2);
833:         smart_str_append(&buf, ":", 1);
834:         smart_str_append(&buf, "f", 1);
835:         smart_str_append(&buf, ":", 1);
836:         smart_str_append(&buf, "i", 2);
837:         smart_str_append(&buf, ":", 1);
838:         smart_str_append(&buf, "f", 1);
839:         smart_str_append(&buf, ":", 1);
840:         smart_str_append(&buf, "i", 2);
841:         smart_str_append(&buf, ":", 1);
842:         smart_str_append(&buf, "f", 1);
843:         smart_str_append(&buf, ":", 1);
844:         smart_str_append(&buf, "i", 2);
845:         smart_str_append(&buf, ":", 1);
846:         smart_str_append(&buf, "f", 1);
847:         smart_str_append(&buf, ":", 1);
848:         smart_str_append(&buf, "i", 2);
849:         smart_str_append(&buf, ":", 1);
850:         smart_str_append(&buf, "f", 1);
851:         smart_str_append(&buf, ":", 1);
852:         smart_str_append(&buf, "i", 2);
853:         smart_str_append(&buf, ":", 1);
854:         smart_str_append(&buf, "f", 1);
855:         smart_str_append(&buf, ":", 1);
856:         smart_str_append(&buf, "i", 2);
857:         smart_str_append(&buf, ":", 1);
858:         smart_str_append(&buf, "f", 1);
859:         smart_str_append(&buf, ":", 1);
860:         smart_str_append(&buf, "i", 2);
861:         smart_str_append(&buf, ":", 1);
862:         smart_str_append(&buf, "f", 1);
863:         smart_str_append(&buf, ":", 1);
864:         smart_str_append(&buf, "i", 2);
865:         smart_str_append(&buf, ":", 1);
866:         smart_str_append(&buf, "f", 1);
867:         smart_str_append(&buf, ":", 1);
868:         smart_str_append(&buf, "i", 2);
869:         smart_str_append(&buf, ":", 1);
870:         smart_str_append(&buf, "f", 1);
871:         smart_str_append(&buf, ":", 1);
872:         smart_str_append(&buf, "i", 2);
873:         smart_str_append(&buf, ":", 1);
874:         smart_str_append(&buf, "f", 1);
875:         smart_str_append(&buf, ":", 1);
876:         smart_str_append(&buf, "i", 2);
877:         smart_str_append(&buf, ":", 1);
878:         smart_str_append(&buf, "f", 1);
879:         smart_str_append(&buf, ":", 1);
880:         smart_str_append(&buf, "i", 2);
881:         smart_str_append(&buf, ":", 1);
882:         smart_str_append(&buf, "f", 1);
883:         smart_str_append(&buf, ":", 1);
884:         smart_str_append(&buf, "i", 2);
885:         smart_str_append(&buf, ":", 1);
886:         smart_str_append(&buf, "f", 1);
887:         smart_str_append(&buf, ":", 1);
888:         smart_str_append(&buf, "i", 2);
889:         smart_str_append(&buf, ":", 1);
890:         smart_str_append(&buf, "f", 1);
891:         smart_str_append(&buf, ":", 1);
892:         smart_str_append(&buf, "i", 2);
893:         smart_str_append(&buf, ":", 1);
894:         smart_str_append(&buf, "f", 1);
895:         smart_str_append(&buf, ":", 1);
896:         smart_str_append(&buf, "i", 2);
897:         smart_str_append(&buf, ":", 1);
898:         smart_str_append(&buf, "f", 1);
899:         smart_str_append(&buf, ":", 1);
900:         smart_str_append(&buf, "i", 2);
901:         smart_str_append(&buf, ":", 1);
902:         smart_str_append(&buf, "f", 1);
903:         smart_str_append(&buf, ":", 1);
904:         smart_str_append(&buf, "i", 2);
905:         smart_str_append(&buf, ":", 1);
906:         smart_str_append(&buf, "f", 1);
907:         smart_str_append(&buf, ":", 1);
908:         smart_str_append(&buf, "i", 2);
909:         smart_str_append(&buf, ":", 1);
910:         smart_str_append(&buf, "f", 1);
911:         smart_str_append(&buf, ":", 1);
912:         smart_str_append(&buf, "i", 2);
913:         smart_str_append(&buf, ":", 1);
914:         smart_str_append(&buf, "f", 1);
915:         smart_str_append(&buf, ":", 1);
916:         smart_str_append(&buf, "i", 2);
917:         smart_str_append(&buf, ":", 1);
918:         smart_str_append(&buf, "f", 1);
919:         smart_str_append(&buf, ":", 1);
920:         smart_str_append(&buf, "i", 2);
921:         smart_str_append(&buf, ":", 1);
922:         smart_str_append(&buf, "f", 1);
923:         smart_str_append(&buf, ":", 1);
924:         smart_str_append(&buf, "i", 2);
925:         smart_str_append(&buf, ":", 1);
926:         smart_str_append(&buf, "f", 1);
927:         smart_str_append(&buf, ":", 1);
928:         smart_str_append(&buf, "i", 2);
929:         smart_str_append(&buf, ":", 1);
930:         smart_str_append(&buf, "f", 1);
931:         smart_str_append(&buf, ":", 1);
932:         smart_str_append(&buf, "i", 2);
933:         smart_str_append(&buf, ":", 1);
934:         smart_str_append(&buf, "f", 1);
935:         smart_str_append(&buf, ":", 1);
936:         smart_str_append(&buf, "i", 2);
937:         smart_str_append(&buf, ":", 1);
938:         smart_str_append(&buf, "f", 1);
939:         smart_str_append(&buf, ":", 1);
940:         smart_str_append(&buf, "i", 2);
941:         smart_str_append(&buf, ":", 1);
942:         smart_str_append(&buf, "f", 1);
943:         smart_str_append(&buf, ":", 1);
944:         smart_str_append(&buf, "i", 2);
945:         smart_str_append(&buf, ":", 1);
946:         smart_str_append(&buf, "f", 1);
947:         smart_str_append(&buf, ":", 1);
948:         smart_str_append(&buf, "i", 2);
949:         smart_str_append(&buf, ":", 1);
950:         smart_str_append(&buf, "f", 1);
951:         smart_str_append(&buf, ":", 1);
952:         smart_str_append(&buf, "i", 2);
953:         smart_str_append(&buf, ":", 1);
954:         smart_str_append(&buf, "f", 1);
955:         smart_str_append(&buf, ":", 1);
956:         smart_str_append(&buf, "i", 2);
957:         smart_str_append(&buf, ":", 1);
958:         smart_str_append(&buf, "f", 1);
959:         smart_str_append(&buf, ":", 1);
960:         smart_str_append(&buf, "i", 2);
961:         smart_str_append(&buf, ":", 1);
962:         smart_str_append(&buf, "f", 1);
963:         smart_str_append(&buf, ":", 1);
964:         smart_str_append(&buf, "i", 2);
965:         smart_str_append(&buf, ":", 1);
966:         smart_str_append(&buf, "f", 1);
967:         smart_str_append(&buf, ":", 1);
968:         smart_str_append(&buf, "i", 2);
969:         smart_str_append(&buf, ":", 1);
970:         smart_str_append(&buf, "f", 1);
971:         smart_str_append(&buf, ":", 1);
972:         smart_str_append(&buf, "i", 2);
973:         smart_str_append(&buf, ":", 1);
974:         smart_str_append(&buf, "f", 1);
975:         smart_str_append(&buf, ":", 1);
976:         smart_str_append(&buf, "i", 2);
977:         smart_str_append(&buf, ":", 1);
978:         smart_str_append(&buf, "f", 1);
979:         smart_str_append(&buf, ":", 1);
980:         smart_str_append(&buf, "i", 2);
981:         smart_str_append(&buf, ":", 1);
982:         smart_str_append(&buf, "f", 1);
983:         smart_str_append(&buf, ":", 1);
984:         smart_str_append(&buf, "i", 2);
985:         smart_str_append(&buf, ":", 1);
986:         smart_str_append(&buf, "f", 1);
987:         smart_str_append(&buf, ":", 1);
988:         smart_str_append(&buf, "i", 2);
989:         smart_str_append(&buf, ":", 1);
990:         smart_str_append(&buf, "f", 1);
991:         smart_str_append(&buf, ":", 1);
992:         smart_str_append(&buf, "i", 2);
993:         smart_str_append(&buf, ":", 1);
994:         smart_str_append(&buf, "f", 1);
995:         smart_str_append(&buf, ":", 1);
996:         smart_str_append(&buf, "i", 2);
997:         smart_str_append(&buf, ":", 1);
998:         smart_str_append(&buf, "f", 1);
999:         smart_str_append(&buf, ":", 1);
1000:         smart_str_append(&buf, "i", 2);
1001:         smart_str_append(&buf, ":", 1);
1002:         smart_str_append(&buf, "f", 1);
1003:         smart_str_append(&buf, ":", 1);
1004:         smart_str_append(&buf, "i", 2);
1005:         smart_str_append(&buf, ":", 1);
1006:         smart_str_append(&buf, "f", 1);
1007:         smart_str_append(&buf, ":", 1);
1008:         smart_str_append(&buf, "i", 2);
1009:         smart_str_append(&buf, ":", 1);
1010:         smart_str_append(&buf, "f", 1);
1011:         smart_str_append(&buf, ":", 1);
1012:         smart_str_append(&buf, "i", 2);
1013:         smart_str_append(&buf, ":", 1);
1014:         smart_str_append(&buf, "f", 1);
1015:         smart_str_append(&buf, ":", 1);
1016:         smart_str_append(&buf, "i", 2);
1017:         smart_str_append(&buf, ":", 1);
1018:         smart_str_append(&buf, "f", 1);
1019:         smart_str_append(&buf, ":", 1);
1020:         smart_str_append(&buf, "i", 2);
1021:         smart_str_append(&buf, ":", 1);
1022:         smart_str_append(&buf, "f", 1);
1023:         smart_str_append(&buf, ":", 1);
1024:         smart_str_append(&buf, "i", 2);
1025:         smart_str_append(&buf, ":", 1);
1026:         smart_str_append(&buf, "f", 1);
1027:         smart_str_append(&buf, ":", 1);
1028:         smart_str_append(&buf, "i", 2);
1029:         smart_str_append(&buf, ":", 1);
1030:         smart_str_append(&buf, "f", 1);
1031:         smart_str_append(&buf, ":", 1);
1032:         smart_str_append(&buf, "i", 2);
1033:         smart_str_append(&buf, ":", 1);
1034:         smart_str_append(&buf, "f", 1);
1035:         smart_str_append(&buf, ":", 1);
1036:         smart_str_append(&buf, "i", 2);
1037:         smart_str_append(&buf, ":", 1);
1038:         smart_str_append(&buf, "f", 1);
1039:         smart_str_append(&buf, ":", 1);
1040:         smart_str_append(&buf, "i", 2);
1041:         smart_str_append(&buf, ":", 1);
1042:         smart_str_append(&buf, "f", 1);
1043:         smart_str_append(&buf, ":", 1);
1044:         smart_str_append(&buf, "i", 2);
1045:         smart_str_append(&buf, ":", 1);
1046:         smart_str_append(&buf, "f", 1);
1047:         smart_str_append(&buf, ":", 1);
1048:         smart_str_append(&buf, "i", 2);
1049:         smart_str_append(&buf, ":", 1);
1050:         smart_str_append(&buf, "f", 1);
1051:         smart_str_append(&buf, ":", 1);
1052:         smart_str_append(&buf, "i", 2);
1053:         smart_str_append(&buf, ":", 1);
1054:         smart_str_append(&buf, "f", 1);
1055:         smart_str_append(&buf, ":", 1);
1056:         smart_str_append(&buf, "i", 2);
1057:         smart_str_append(&buf, ":", 1);
1058:         smart_str_append(&buf, "f", 1);
1059:         smart_str_append(&buf, ":", 1);
1060:         smart_str_append(&buf, "i", 2);
1061:         smart_str_append(&buf, ":", 1);
1062:         smart_str_append(&buf, "f", 1);
1063:         smart_str_append(&buf, ":", 1);
1064:         smart_str_append(&buf, "i", 2);
1065:         smart_str_append(&buf, ":", 1);
1066:         smart_str_append(&buf, "f", 1);
1067:         smart_str_append(&buf, ":", 1);
1068:         smart_str_append(&buf, "i", 2);
1069:         smart_str_append(&buf, ":", 1);
1070:         smart_str_append(&buf, "f", 1);
1071:         smart_str_append(&buf, ":", 1);
1072:         smart_str_append(&buf, "i", 2);
1073:         smart_str_append(&buf, ":", 1);
1074:         smart_str_append(&buf, "f", 1);
1075:         smart_str_append(&buf, ":", 1);
1076:         smart_str_append(&buf, "i", 2);
1077:         smart_str_append(&buf, ":", 1);
1078:         smart_str_append(&buf, "f", 1);
1079:         smart_str_append(&buf, ":", 1);
1080:         smart_str_append(&buf, "i", 2);
1081:         smart_str_append(&buf, ":", 1);
1082:         smart_str_append(&buf, "f", 1);
1083:         smart_str_append(&buf, ":", 1);
1084:         smart_str_append(&buf, "i", 2);
1085:         smart_str_append(&buf, ":", 1);
1086:         smart_str_append(&buf, "f", 1);
1087:         smart_str_append(&buf, ":", 1);
1088:         smart_str_append(&buf, "i", 2);
1089:         smart_str_append(&buf, ":", 1);
1090:         smart_str_append(&buf, "f", 1);
1091:         smart_str_append(&buf, ":", 1);
1092:         smart_str_append(&buf, "i", 2);
1093:         smart_str_append(&buf, ":", 1);
1094:         smart_str_append(&buf, "f", 1);
1095:         smart_str_append(&buf, ":", 1);
1096:         smart_str_append(&buf, "i", 2);
1097:         smart_str_append(&buf, ":", 1);
1098:         smart_str_append(&buf, "f", 1);
1099:         smart_str_append(&buf, ":", 1);
1100:         smart_str_append(&buf, "i", 2);
1101:         smart_str_append(&buf, ":", 1);
1102:         smart_str_append(&buf, "f", 1);
1103:         smart_str_append(&buf, ":", 1);
1104:         smart_str_append(&buf, "i", 2);
1105:         smart_str_append(&buf, ":", 1);
1106:         smart_str_append(&buf, "f", 1);
1107:         smart_str_append(&buf, ":", 1);
1108:         smart_str_append(&buf, "i", 2);
1109:         smart_str_append(&buf, ":", 1);
1110:         smart_str_append(&amp
```

```

753: SPL_METHOD(splObjectStorage, unserialize)
754: {
755:     splObjectStorage *intern = Z_SPLOBJ_STORAGE_P(getThis());
756:
757:     char *buf;
758:     size_t buf_len;
759:     const unsigned char *p;
760:     php_unserialize_data_t var_hash;
761:     zval entry, info;
762:     zval *pcount, *pmembers;
763:     spl_SplObjectStorageElement *element;
764:     zend_long count;
765:
766:     if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "s", &buf, &buf_len) == FAILURE) {
767:         return;
768:     }
769:
770:     if (buf_len == 0) {
771:         return;
772:     }
773:
774:     /* storage */
775:     s = p = (const unsigned char*)buf;
776:     PHP_VAR_UNSERIALIZE_INIT(var_hash);
777:
778:     if (*p != 'p' || **p != ':' || *++p != ':') {
779:         goto outexcept;
780:     }
781:     **p;
782:
783:     pcount = var_tmp_var(&var_hash);
784:     if (!php_var_unserialize(pcount, &p, s + buf_len, &var_hash) || Z_TYPE_P(pcount) != IS_LONG) {
785:         goto outexcept;
786:     }
787:
788:     --p; /* for 'p:' */
789:     count = Z_LVAL_P(pcount);
790:
791:     ZVAL_UNDEF(&entry);
792:     ZVAL_UNDEF(&info);
793:
794:     while (count-- > 0) {
795:         spl_SplObjectStorageElement *element;
796:         zend_hash_key key;
797:
798:         if (*p != ':' || *++p != ':') {
799:             goto outexcept;
800:         }
801:         **p;
802:         if (*p != 'O' || **p != 'C' || *++p != 'C' || *++p != 'r') {
803:             goto outexcept;
804:         }
805:         /* store reference to allow cross-references between different elements */
806:         if (!php_var_unserialize(&entry, &p, s + buf_len, &var_hash)) {
807:             goto outexcept;
808:         }
809:         if (*p == ',' || /* new version has inf */)
810:             **p;
811:         if (!php_var_unserialize(&info, &p, s + buf_len, &var_hash)) {
812:             goto outexcept;
813:         }
814:
815:         if (Z_TYPE(entry) != IS_OBJECT) {
816:             zval_ptr_dtor(&entry);
817:             zval_ptr_dtor(&info);
818:             goto outexcept;
819:         }
820:
821:         if (spl_object_storage_get_hash(&key, intern, getThis(), &entry) == FAILURE) {
822:             zval_ptr_dtor(&entry);
823:             zval_ptr_dtor(&info);
824:             goto outexcept;
825:         }
826:
827:         element = spl_object_storage_get(intern, &key);
828:         spl_object_storage_free_hash(intern, &key);
829:         if (element) {
830:             if (!Z_UNDEF(element->info)) {
831:                 var_push_dtor(&var_hash, &element->info);
832:             }
833:             if (!Z_UNDEF(element->obj)) {
834:                 var_push_dtor(&var_hash, &element->obj);
835:             }
836:         }
837:         element = spl_object_storage_attach(intern, getThis(), &entry, Z_UNDEF(&info) ? NULL : &info);
838:         var_replace(&var_hash, &entry, &element->obj);
839:         var_replace(&var_hash, &info, &element->info);
840:         zval_ptr_dtor(&entry);
841:         ZVAL_UNDEF(&entry);
842:         zval_ptr_dtor(&info);
843:         ZVAL_UNDEF(&info);
844:     }
845:
846:     if (*p != ':' || *++p != ':') {
847:         goto outexcept;
848:     }
849:     **p;
850:
851:     /* members */
852:     if (*p != 'm' || **p != ':' || *++p != ':') {
853:         goto outexcept;
854:     }
855:     **p;
856:
857:     pmembers = var_tmp_var(&var_hash);
858:     if (!php_var_unserialize(pmembers, &p, s + buf_len, &var_hash) || Z_TYPE_P(pmembers) != IS_ARRAY) {
859:         goto outexcept;
860:     }
861:
862:     /* copy members */
863:     object_properties_load(&intern->std, Z_ARRVAL_P(pmembers));
864:
865:     PHP_VAR_UNSERIALIZE_DESTROY(&var_hash);
866:     return;
867:
868: outexcept:
869:     PHP_VAR_UNSERIALIZE_DESTROY(&var_hash);
870:     zend_throw_exception_ex(spl_ce_UnexpectedValueException, 0, "Error at offset %d of %d bytes", ((char*)p - buf), buf_len);
871:     return;
872:
873:     /* }}} */
874:
875: ZEND_BEGIN_ARG_INFO(arginfo_Object, 0)
876:     ZEND_ARG_INFO(0, object)
877: ZEND_END_ARG_INFO()
878:
879: ZEND_BEGIN_ARG_INFO_EX(arginfo_Attach, 0, 0, 1)
880:     ZEND_ARG_INFO(0, object)
881:     ZEND_ARG_INFO(0, info)
882: ZEND_END_ARG_INFO()
883:
884: ZEND_BEGIN_ARG_INFO(arginfo_Serialize, 0)
885:     ZEND_ARG_INFO(0, serialized)
886: ZEND_END_ARG_INFO()
887:
888: ZEND_BEGIN_ARG_INFO(arginfo_GetInfo, 0)
889:     ZEND_ARG_INFO(0, info)
890: ZEND_END_ARG_INFO()
891:
892: ZEND_BEGIN_ARG_INFO(arginfo_GetHash, 0)
893:     ZEND_ARG_INFO(0, object)
894: ZEND_END_ARG_INFO()
895:
896: ZEND_BEGIN_ARG_INFO_EX(arginfo_OffsetGet, 0, 0, 1)
897:     ZEND_ARG_INFO(0, object)
898: ZEND_END_ARG_INFO()
899:
900: ZEND_BEGIN_ARG_INFO(arginfo_SplObjectVoid, 0)
901: ZEND_END_ARG_INFO()
902:
903: static const zend_function_entry spl_funcs_splObjectStorage[] = {
904:     SPL_ME(splObjectStorage, attach, arginfo_attach, 0),
905:     SPL_ME(splObjectStorage, detach, arginfo_detach, 0),
906:     SPL_ME(splObjectStorage, contains, arginfo_contains, 0),
907:     SPL_ME(splObjectStorage, addAll, arginfo_addAll, 0),
908:     SPL_ME(splObjectStorage, removeAll, arginfo_removeAll, 0),
909:     SPL_ME(splObjectStorage, removeAllExcept, arginfo_removeAllExcept, 0),
910:     SPL_ME(splObjectStorage, getInfo, arginfo_splObjectVoid, 0),
911:     SPL_ME(splObjectStorage, setInfo, arginfo_setInfo, 0),
912:     SPL_ME(splObjectStorage, getHash, arginfo_getHash, 0),
913:     /* Constant */
914:     SPL_ME(splObjectStorage, count, arginfo_splObjectVoid, 0),
915:     /* Iterator */
916:     SPL_ME(splObjectStorage, rewind, arginfo_splObjectVoid, 0),
917:     SPL_ME(splObjectStorage, valid, arginfo_splObjectVoid, 0),
918:     SPL_ME(splObjectStorage, key, arginfo_splObjectVoid, 0),
919:     SPL_ME(splObjectStorage, current, arginfo_splObjectVoid, 0),
920:     SPL_ME(splObjectStorage, next, arginfo_splObjectVoid, 0),
921:     /* Serializable */
922:     SPL_ME(splObjectStorage, unserialize, arginfo_Serialize, 0),
923:     SPL_ME(splObjectStorage, serialize, arginfo_splObjectVoid, 0),
924:     /* ArrayAccess */
925:     SPL_MA(splObjectStorage, offsetExists, splObjectStorage, contains, arginfo_OffsetGet, 0),
926:     SPL_MA(splObjectStorage, offsetSet, splObjectStorage, attach, arginfo_attach, 0),
927:     SPL_MA(splObjectStorage, offsetUnset, splObjectStorage, detach, arginfo_OffsetGet, 0),
928:     SPL_ME(splObjectStorage, offsetGet, arginfo_OffsetGet, 0),
929:     PHP_FE_END
930: };
931:
932: typedef enum {
933:     MIT_NEED_ANY = 0,
934:     MIT_NEED_ALL = 1,
935:     MIT_KEYS_NUMERIC = 0,
936:     MIT_KEYS_ASSOC = 2
937: } MultiplierIteratorFlags;
938:
939: #define SPL_MULTIPLIER_ITERATOR_GET_ALL_CW 1
940: #define SPL_MULTIPLIER_ITERATOR_GET_ALL_KEY 2

```

```

941:
942: /* {{{ proto void MultiplierIterator::__construct([int flags = MIT_NEED_ALL|MIT_KEYS_NUMERIC])
943:    Iterator that iterates over several iterators one after the other */
944: SPL_METHOD(MultiplierIterator, __construct)
945: {
946:     spl_SplObjectStorage *intern;
947:     zend_long flags = MIT_NEED_ALL|MIT_KEYS_NUMERIC;
948:
949:     if (zend_parse_parameters_throw(ZEND_NUM_ARGS(), "i", &flags) == FAILURE) {
950:         return;
951:     }
952:
953:     intern = Z_SPLOBJ_STORAGE_P(getThis());
954:     intern->flags = flags;
955: }
956: /* }}} */
957:
958: /* {{{ proto int MultiplierIterator::getFlags()
959:    Return current flags */
960: SPL_METHOD(MultiplierIterator, getFlags)
961: {
962:     spl_SplObjectStorage *intern = Z_SPLOBJ_STORAGE_P(getThis());
963:
964:     if (zend_parse_parameters_none() == FAILURE) {
965:         return;
966:     }
967:     RETURN_LONG(intern->flags);
968: }
969: /* }}} */
970:
971: /* {{{ proto int MultiplierIterator::setFlags(int flags)
972:    Set flags */
973: SPL_METHOD(MultiplierIterator, setFlags)
974: {
975:     spl_SplObjectStorage *intern;
976:     intern = Z_SPLOBJ_STORAGE_P(getThis());
977:
978:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "i", &intern->flags) == FAILURE) {
979:         return;
980:     }
981: }
982: /* }}} */
983:
984: /* {{{ proto void attachIterator(Iterator iterator[, mixed info]) throws InvalidArgumentException
985:    Attach a new iterator */
986: SPL_METHOD(MultiplierIterator, attachIterator)
987: {
988:     spl_SplObjectStorage *intern;
989:     zval *iterator = NULL, *info = NULL;
990:
991:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "Ois", &iterator, &zend_obj_iterator, &info) == FAILURE) {
992:         return;
993:     }
994:
995:     intern = Z_SPLOBJ_STORAGE_P(getThis());
996:
997:     if (info != NULL) {
998:         spl_SplObjectStorageElement *element;
999:
1000:         if (Z_TYPE_P(info) != IS_LONG || Z_TYPE_P(info) != IS_STRING) {
1001:             zend_throw_exception(spl_ce_InvalidArgumentException, "Info must be NULL, integer or string", 0);
1002:             return;
1003:         }
1004:
1005:         zend_hash_internal_pointer_reset_ex(&intern->storage, &intern->pos);
1006:         while (element = zend_hash_get_current_data_ptr_ex(&intern->storage, &intern->pos) != NULL) {
1007:             if (fast_is_identical_function(info, element->info)) {
1008:                 zend_throw_exception(spl_ce_InvalidArgumentException, "Key duplication error", 0);
1009:                 return;
1010:             }
1011:             zend_hash_move_forward_ex(&intern->storage, &intern->pos);
1012:         }
1013:     }
1014:
1015:     spl_obj_iterator_attach(intern, getThis(), iterator, info);
1016: }
1017: /* }}} */
1018:
1019: /* {{{ proto void MultiplierIterator::rewind()
1020:    Rewind all attached iterator instances */
1021: SPL_METHOD(MultiplierIterator, rewind)
1022: {
1023:     spl_SplObjectStorage *intern;
1024:     spl_SplObjectStorageElement *element;
1025:     zval *it;
1026:
1027:     intern = Z_SPLOBJ_STORAGE_P(getThis());
1028:
1029:     if (zend_parse_parameters_none() == FAILURE) {
1030:         return;
1031:     }
1032:
1033:     zend_hash_internal_pointer_reset_ex(&intern->storage, &intern->pos);
1034:     while (element = zend_hash_get_current_data_ptr_ex(&intern->storage, &intern->pos) != NULL || IS_EXCEPTION()) {
1035:         if (element->obj) {
1036:             zend_call_method_with_0_params(it, Z_OBJCE_P(it), &Z_OBJCE_P(it)->iterator_funcs, ZEND_FUNC_REWIND, NULL);
1037:             zend_hash_move_forward_ex(&intern->storage, &intern->pos);
1038:         }
1039:     }
1040: }
1041: /* }}} */
1042:
1043: /* {{{ proto void MultiplierIterator::next()
1044:    Move all attached iterator instances forward */
1045: SPL_METHOD(MultiplierIterator, next)
1046: {
1047:     spl_SplObjectStorage *intern;
1048:     spl_SplObjectStorageElement *element;
1049:     zval *it;
1050:
1051:     intern = Z_SPLOBJ_STORAGE_P(getThis());
1052:
1053:     if (zend_parse_parameters_none() == FAILURE) {
1054:         return;
1055:     }
1056:
1057:     zend_hash_internal_pointer_reset_ex(&intern->storage, &intern->pos);
1058:     while (element = zend_hash_get_current_data_ptr_ex(&intern->storage, &intern->pos) != NULL || IS_EXCEPTION()) {
1059:         if (element->obj) {
1060:             zend_call_method_with_0_params(it, Z_OBJCE_P(it), &Z_OBJCE_P(it)->iterator_funcs, ZEND_FUNC_NEXT, NULL);
1061:             zend_hash_move_forward_ex(&intern->storage, &intern->pos);
1062:         }
1063:     }
1064: }
1065: /* }}} */
1066:
1067: /* {{{ proto bool MultiplierIterator::valid()
1068:    Return whether all or one sub iterator is valid depending on flags */
1069: SPL_METHOD(MultiplierIterator, valid)
1070: {
1071:     spl_SplObjectStorage *intern;
1072:     zval *it, *retval;
1073:     zend_long expect, valid;
1074:
1075:     intern = Z_SPLOBJ_STORAGE_P(getThis());
1076:
1077:     if (zend_parse_parameters_none() == FAILURE) {
1078:         return;
1079:     }
1080:
1081:     if (!zend_hash_num_elements(&intern->storage)) {
1082:         RETURN_FALSE;
1083:     }
1084:
1085:     expect = (intern->flags & MIT_NEED_ALL) ? 1 : 0;
1086:
1087:     zend_hash_internal_pointer_reset_ex(&intern->storage, &intern->pos);
1088:     while (element = zend_hash_get_current_data_ptr_ex(&intern->storage, &intern->pos) != NULL || IS_EXCEPTION()) {
1089:         if (element->obj) {
1090:             zend_call_method_with_0_params(it, Z_OBJCE_P(it), &Z_OBJCE_P(it)->iterator_funcs, ZEND_FUNC_VALID, &retval);
1091:         }
1092:         if (!Z_UNDEF(retval)) {
1093:             valid = (Z_TYPE(retval) == IS_TRUE);
1094:             zval_ptr_dtor(&retval);
1095:         } else {
1096:             valid = 0;
1097:         }
1098:         if (expect != valid) {
1099:             RETURN_BOOL(!expect);
1100:         }
1101:     }
1102:
1103:     zend_hash_move_forward_ex(&intern->storage, &intern->pos);
1104:
1105:     RETURN_BOOL(expect);
1106: }
1107: /* }}} */
1108:
1109: static void spl_multiple_iterator_get_all(spl_SplObjectStorage *intern, int get_type, zval *return_value) /* {{{ */
1110: {
1111:     spl_SplObjectStorageElement *element;
1112:     zval *it, *retval;
1113:     int valid = 1, num_elements;
1114:
1115:     num_elements = zend_hash_num_elements(&intern->storage);
1116:     if (num_elements < 1) {
1117:         RETURN_FALSE;
1118:     }
1119:
1120:     array_init_size(return_value, num_elements);
1121:
1122:     zend_hash_internal_pointer_reset_ex(&intern->storage, &intern->pos);
1123:     while (element = zend_hash_get_current_data_ptr_ex(&intern->storage, &intern->pos) != NULL || IS_EXCEPTION()) {
1124:         if (element->obj) {
1125:             zend_call_method_with_0_params(it, Z_OBJCE_P(it), &Z_OBJCE_P(it)->iterator_funcs, ZEND_FUNC_VALID, &retval);
1126:         }
1127:         if (!Z_UNDEF(retval)) {
1128:             valid = (Z_TYPE(retval) == IS_TRUE);

```



```

1129:     rval_get_dtor(&retval);
1130: } else {
1131:     valid = 0;
1132: }
1133:
1134: if (valid) {
1135:     if (SPL_MULTIPLE_ITERATOR_GET_ALL_CURRENT == get_type) {
1136:         send_call_method_with_0_params(it, z_OBJCE_P(it), z_OBJCE_P(it) -> iterator_funcs.if_current, "current", &retval);
1137:     } else {
1138:         send_call_method_with_0_params(it, z_OBJCE_P(it), z_OBJCE_P(it) -> iterator_funcs.if_key, "key", &retval);
1139:     }
1140:     if (!Z_ISUNDEF(retval)) {
1141:         zend_throw_exception(spl_ce_RuntimeException, "Failed to call sub iterator method", 0);
1142:         return;
1143:     }
1144: } else if (intern->flags & MIT_NEED_ALL) {
1145:     if (SPL_MULTIPLE_ITERATOR_GET_ALL_CURRENT == get_type) {
1146:         zend_throw_exception(spl_ce_RuntimeException, "Called current() with non valid sub iterator", 0);
1147:     } else {
1148:         zend_throw_exception(spl_ce_RuntimeException, "Called key() with non valid sub iterator", 0);
1149:     }
1150:     return;
1151: } else {
1152:     ZVAL_NULL(&retval);
1153: }
1154:
1155: if (intern->flags & MIT_KEYS_ASSOC) {
1156:     switch (z_TYPE(element->info)) {
1157:         case IS_LONG:
1158:             add_index_zval(return_value, z_IVAL(element->info), &retval);
1159:             break;
1160:         case IS_STRING:
1161:             zend_symtable_update(z_ARRVAL_P(return_value), z_STR(element->info), &retval);
1162:             break;
1163:         default:
1164:             rval_get_dtor(&retval);
1165:             zend_throw_exception(spl_ce_InvalidArgumentException, "Sub-Iterator is associated with NULL", 0);
1166:             return;
1167:     }
1168: } else {
1169:     add_next_index_zval(return_value, &retval);
1170: }
1171:
1172: zend_hash_move_forward_ex(&intern->storage, &intern->pos);
1173: }
1174: /* }}} */
1175: /* }}} */
1176:
1177: /* {{{ proto array current() throws RuntimeException throws InvalidArgumentException
1178:    Return an array of all registered Iterator instances current() result */
1179: SPL_METHOD(MultipleIterator, current)
1180: {
1181:     spl_SplObjectStorage *intern;
1182:     intern = z_SPL_OBJECT_STORAGE_P(getThis());
1183:
1184:     if (zend_parse_parameters_none() == FAILURE) {
1185:         return;
1186:     }
1187:
1188:     spl_multiple_iterator_get_all(intern, SPL_MULTIPLE_ITERATOR_GET_ALL_CURRENT, return_value);
1189: }
1190: /* }}} */
1191:
1192: /* {{{ proto array MultipleIterator::key()
1193:    Return an array of all registered Iterator instances key() result */
1194: SPL_METHOD(MultipleIterator, key)
1195: {
1196:     spl_SplObjectStorage *intern;
1197:     intern = z_SPL_OBJECT_STORAGE_P(getThis());
1198:
1199:     if (zend_parse_parameters_none() == FAILURE) {
1200:         return;
1201:     }
1202:
1203:     spl_multiple_iterator_get_all(intern, SPL_MULTIPLE_ITERATOR_GET_ALL_KEY, return_value);
1204: }
1205: /* }}} */
1206:
1207: ZEND_BEGIN_ARG_INFO_EX(arginfo_MultipleIterator_attachIterator, 0, 0, 1)
1208:     ZEND_ARG_OBJ_INFO(0, iterator, Iterator, 0)
1209: ZEND_END_ARG_INFO();
1210: ZEND_FUNC_INFO();
1211:
1212: ZEND_BEGIN_ARG_INFO_EX(arginfo_MultipleIterator_detachIterator, 0, 0, 1)
1213:     ZEND_ARG_OBJ_INFO(0, iterator, Iterator, 0)
1214: ZEND_END_ARG_INFO();
1215:
1216: ZEND_BEGIN_ARG_INFO_EX(arginfo_MultipleIterator_containsIterator, 0, 0, 1)
1217:     ZEND_ARG_OBJ_INFO(0, iterator, Iterator, 0)
1218: ZEND_END_ARG_INFO();
1219:
1220: ZEND_BEGIN_ARG_INFO_EX(arginfo_MultipleIterator_setFlags, 0, 0, 1)
1221:     ZEND_ARG_INFO(0, flags)
1222: ZEND_END_ARG_INFO();
1223:
1224: static const zend_function_entry spl_funcs_MultipleIterator[] = {
1225:     SPL_ME(MultipleIterator, __construct,      arginfo_MultipleIterator_setFlags,      0)
1226:     SPL_ME(MultipleIterator, getFlags,         arginfo_splObject_void,                0)
1227:     SPL_ME(MultipleIterator, setFlags,         arginfo_MultipleIterator_setFlags,     0)
1228:     SPL_ME(MultipleIterator, attachIterator,   arginfo_MultipleIterator_attachIterator, 0)
1229:     SPL_MA(MultipleIterator, detachIterator,   splObjectStorage, detach,      arginfo_MultipleIterator_detachIterator, 0)
1230:     SPL_MA(MultipleIterator, containsIterator, splObjectStorage, contains,    arginfo_MultipleIterator_containsIterator, 0)
1231:     SPL_MA(MultipleIterator, countIterators,  splObjectStorage, count,      arginfo_splObject_void,                0)
1232:     /* Iterator */
1233:     SPL_ME(MultipleIterator, rewind,           arginfo_splObject_void,                0)
1234:     SPL_ME(MultipleIterator, valid,           arginfo_splObject_void,                0)
1235:     SPL_ME(MultipleIterator, key,             arginfo_splObject_void,                0)
1236:     SPL_ME(MultipleIterator, current,         arginfo_splObject_void,                0)
1237:     SPL_ME(MultipleIterator, next,            arginfo_splObject_void,                0)
1238:     PHP_FE_END
1239: };
1240:
1241: /* {{{ PHP_MINIT_FUNCTION(spl_observer) */
1242: PHP_MINIT_FUNCTION(spl_observer)
1243: {
1244:     REGISTER_SPL_INTERFACE(splObserver);
1245:     REGISTER_SPL_INTERFACE(splSubject);
1246:
1247:     REGISTER_SPL_STD_CLASS_EX(splObjectStorage, spl_SplObjectStorage_new, spl_funcs_splObjectStorage);
1248:     memcpy(&spl_handler_splObjectStorage, zend_get_std_object_handlers(), sizeof(zend_object_handlers));
1249:
1250:     spl_handler_splObjectStorage.offset = XOffsetOf(spl_SplObjectStorage, std);
1251:     spl_handler_splObjectStorage.get_debug_info = spl_object_storage_debug_info;
1252:     spl_handler_splObjectStorage.compare_objects = spl_object_storage_compare_objects;
1253:     spl_handler_splObjectStorage.clone_obj = spl_object_storage_clone;
1254:     spl_handler_splObjectStorage.get_gc = spl_object_storage_get_gc;
1255:     spl_handler_splObjectStorage.dtor_obj = zend_object_dtor_obj;
1256:     spl_handler_splObjectStorage.free_obj = spl_SplObjectStorage_free_storage;
1257:
1258:     REGISTER_SPL_IMPLEMENTATIONS(splObjectStorage, Countable);
1259:     REGISTER_SPL_IMPLEMENTATIONS(splObjectStorage, Iterator);
1260:     REGISTER_SPL_IMPLEMENTATIONS(splObjectStorage, Serializable);
1261:     REGISTER_SPL_IMPLEMENTATIONS(splObjectStorage, ArrayAccess);
1262:
1263:     REGISTER_SPL_STD_CLASS_EX(MultipleIterator, spl_SplObjectStorage_new, spl_funcs_MultipleIterator);
1264:     REGISTER_SPL_ITERATOR(MultipleIterator);
1265:
1266:     REGISTER_SPL_CLASS_CONST_LONG(MultipleIterator, "MIT_NEED_ANY", MIT_NEED_ANY);
1267:     REGISTER_SPL_CLASS_CONST_LONG(MultipleIterator, "MIT_NEED_ALL", MIT_NEED_ALL);
1268:     REGISTER_SPL_CLASS_CONST_LONG(MultipleIterator, "MIT_KEYS_NUMERIC", MIT_KEYS_NUMERIC);
1269:     REGISTER_SPL_CLASS_CONST_LONG(MultipleIterator, "MIT_KEYS_ASSOC", MIT_KEYS_ASSOC);
1270:
1271:     return SUCCESS;
1272: }
1273: /* }}} */
1274:
1275: /*
1276:  * Local variables:
1277:  * tab-width: 4
1278:  * c-basic-offset: 4
1279:  * End:
1280:  * vim600: fdm=marker
1281:  * vim: noet sw=4 ts=4
1282:  */

```

```

1: /*
2:  * =====
3:  * ( PHP Version ? )
4:  * =====
5:  * Copyright (c) 1997-2018 The PHP Group
6:  * =====
7:  * This source file is subject to version 3.01 of the PHP license,
8:  * that is bundled with this package in the file LICENSE, and is
9:  * available through the world-wide-web at the following url:
10:  * http://www.php.net/license/3.01.txt
11:  * If you did not receive a copy of the PHP license and are unable to
12:  * obtain it through the world-wide-web, please send a note to
13:  * license@php.net so we can mail you a copy immediately.
14:  * =====
15:  * Authors: Etienne Kneuss <ekd@php.net>
16:  * =====
17:  */
18:
19: /* $Id$ */
20:
21: #ifndef HAVE_CONFIG_H
22: #include "config.h"
23: #endif
24:
25: #include "php.h"
26: #include "zend_exceptions.h"
27: #include "zend_hash.h"
28:
29: #include "php_spl.h"
30: #include "ext/standard/info.h"
31: #include "ext/standard/php_var.h"
32: #include "zend_smart_str.h"
33: #include "spl_functions.h"
34: #include "spl_weakmap.h"
35: #include "spl_iterators.h"
36: #include "spl_dlist.h"
37: #include "spl_exceptions.h"
38:
39: zend_object_handlers spl_handler_spldoublylinkedlist;
40: PHPAPI zend_class_entry *spl_ce_spldoublylinkedlist;
41: PHPAPI zend_class_entry *spl_ce_splQueue;
42: PHPAPI zend_class_entry *spl_ce_splStack;
43:
44: #define SPL_LLIST_DELEAF(elem) if (t--(elem)-->rc) { \
45:     efree(elem); \
46: }
47:
48: #define SPL_LLIST_CHECK_DELEAF(elem) if ((elem) && t--(elem)-->rc) { \
49:     efree(elem); \
50: }
51:
52: #define SPL_LLIST_ADDREF(elem) (elem)-->rc++
53: #define SPL_LLIST_CHECK_ADDREF(elem) if (elem) (elem)-->rc++
54:
55: #define SPL_LLIST_IT_DELEAF 0x00000001 /* Delete flag makes the iterator delete the current element on next */
56: #define SPL_LLIST_IT_LIFO 0x00000002 /* LIFO flag makes the iterator traverse the structure as a lastInFirstOut */
57: #define SPL_LLIST_IT_MASK 0x00000003 /* Mask to isolate flags related to iterators */
58: #define SPL_LLIST_IT_FIX 0x00000004 /* Backward/Forward bit is fixed */
59:
60: #ifndef ACCEPT
61: #define ACCEPT
62: #endif
63:
64: typedef struct _spl_ptr_llist_element {
65:     struct _spl_ptr_llist_element *prev;
66:     struct _spl_ptr_llist_element *next;
67:     int rc;
68:     zval data;
69: } spl_ptr_llist_element;
70:
71: typedef void (*spl_ptr_llist_dtor_func)(spl_ptr_llist_element *);
72: typedef void (*spl_ptr_llist_ctor_func)(spl_ptr_llist_element *);
73:
74: typedef struct _spl_ptr_llist {
75:     spl_ptr_llist_element *head;
76:     spl_ptr_llist_element *tail;
77:     spl_ptr_llist_dtor_func dtor;
78:     spl_ptr_llist_ctor_func ctor;
79:     int count;
80: } spl_ptr_llist;
81:
82: typedef struct _spl_dlist_object spl_dlist_object;
83: typedef struct _spl_dlist_iterator spl_dlist_iterator;
84:
85: struct _spl_dlist_object {
86:     spl_ptr_llist *llist;
87:     int traverse_position;
88:     spl_ptr_llist_element *traverse_pointer;
89:     int flags;
90:     zend_function *repr_offset_get;
91:     zend_function *repr_offset_set;
92:     zend_function *repr_offset_del;
93:     zend_function *repr_offset_has;
94:     zend_function *repr_offset_del;
95:     zend_function *repr_count;
96:     zend_function *repr_get_iterator;
97:     zval *gc_data;
98:     int gc_data_count;
99:     zend_object std;
100: };
101:
102: /* Define an overloaded iterator structure */
103: struct _spl_dlist_iterator {
104:     zend_object_iterator intern;
105:     spl_ptr_llist_element *traverse_pointer;
106:     int traverse_position;
107:     int flags;
108: };
109:
110: static inline spl_dlist_object *spl_dlist_from_obj(zend_object *obj) /* {{{ */ {
111:     return (spl_dlist_object*)(char*)(obj) - sizeof(spl_dlist_object, std);
112: } /* }}} */
113:
114: #define SPL_DLIST_P(rv) spl_dlist_from_obj(Z_OBJ_P(rv))
115:
116: /* {{{ spl_ptr_llist */
117: static void spl_ptr_llist_val_dtor(spl_ptr_llist_element *elem) /* {{{ */ {
118:     if (!Z_ISMMAGIC(elem->data)) {
119:         spl_ptr_llist_dtor_func dtor;
120:         ZVAL_UNDEF(elem->data);
121:     }
122: }
123: /* }}} */
124:
125: static void spl_ptr_llist_val_ctor(spl_ptr_llist_element *elem) /* {{{ */ {
126:     if (!Z_ISMMAGIC(elem->data)) {
127:         Z_ADDREF(elem->data);
128:     }
129: }
130: /* }}} */
131:
132: static spl_ptr_llist *spl_ptr_llist_init(spl_ptr_llist_ctor_func ctor, spl_ptr_llist_dtor_func dtor) /* {{{ */ {
133:     spl_ptr_llist *llist = emalloc(sizeof(spl_ptr_llist));
134:     llist->head = NULL;
135:     llist->tail = NULL;
136:     llist->count = 0;
137:     llist->dtor = dtor;
138:     llist->ctor = ctor;
139:     return llist;
140: } /* }}} */
141:
142: static zend_long spl_ptr_llist_count(spl_ptr_llist *llist) /* {{{ */ {
143:     return llist->count;
144: } /* }}} */
145:
146: static void spl_ptr_llist_destroy(spl_ptr_llist *llist) /* {{{ */ {
147:     spl_ptr_llist_element *current = llist->head, *next;
148:     spl_ptr_llist_dtor_func dtor = llist->dtor;
149:     while (current) {
150:         next = current->next;
151:         if (dtor) {
152:             dtor(current);
153:         }
154:         SPL_LLIST_DELEAF(current);
155:         current = next;
156:     }
157:     efree(llist);
158: } /* }}} */
159:
160: static spl_ptr_llist_element *spl_ptr_llist_offset(spl_ptr_llist *llist, zend_long offset, int backward) /* {{{ */ {
161:     spl_ptr_llist_element *current;
162:     int pos = 0;
163:     if (backward) {
164:         current = llist->tail;
165:     } else {
166:         current = llist->head;
167:     }
168:     while (current && pos < offset) {
169:         pos++;
170:         if (backward) {
171:             current = current->prev;
172:         } else {
173:             current = current->next;
174:         }
175:     }
176:     return current;
177: } /* }}} */
178:
179: static void spl_ptr_llist_unshift(spl_ptr_llist *llist, zval *data) /* {{{ */ {
180:     spl_ptr_llist_element *elem = emalloc(sizeof(spl_ptr_llist_element));
181:     elem->rc = 1;
182:     elem->prev = NULL;
183:     elem->next = llist->head;
184:     ZVAL_COPY_VALUE(elem->data, data);
185:     if (llist->head) {
186:         llist->head->prev = elem;
187:     }
188:     llist->head = elem;
189:     llist->count++;
190:     if (llist->dtor) {
191:         llist->dtor(elem);
192:     }
193: } /* }}} */
194:
195: static void spl_ptr_llist_push(spl_ptr_llist *llist, zval *data) /* {{{ */ {
196:     spl_ptr_llist_element *elem = emalloc(sizeof(spl_ptr_llist_element));
197:     elem->rc = 1;
198:     elem->prev = llist->tail;
199:     elem->next = NULL;
200:     ZVAL_COPY_VALUE(elem->data, data);
201:     if (llist->tail) {
202:         llist->tail->next = elem;
203:     }
204:     llist->tail = elem;
205:     llist->count++;
206:     if (llist->dtor) {
207:         llist->dtor(elem);
208:     }
209: } /* }}} */
210:
211: static void spl_ptr_llist_pop(spl_ptr_llist *llist, zval *ret) /* {{{ */ {
212:     spl_ptr_llist_element *tail = llist->tail;
213:     if (tail == NULL) {
214:         return;
215:     }
216:     ZVAL_UNDEF(ret);
217:     return;
218: } /* }}} */
219:
220: static void spl_ptr_llist_last(spl_ptr_llist *llist, zval *ret) /* {{{ */ {
221:     spl_ptr_llist_element *tail = llist->tail;
222:     if (tail == NULL) {
223:         return;
224:     }
225:     ZVAL_COPY(ret, tail->data);
226:     return;
227: } /* }}} */
228:
229: static void spl_ptr_llist_first(spl_ptr_llist *llist, zval *ret) /* {{{ */ {
230:     spl_ptr_llist_element *head = llist->head;
231:     if (head == NULL) {
232:         return;
233:     }
234:     ZVAL_COPY(ret, head->data);
235:     return;
236: } /* }}} */
237:
238: static void spl_ptr_llist_shift(spl_ptr_llist *llist, zval *ret) /* {{{ */ {
239:     spl_ptr_llist_element *head = llist->head;
240:     if (head == NULL) {
241:         return;
242:     }
243:     ZVAL_COPY(ret, head->data);
244:     SPL_LLIST_DELEAF(head);
245:     return;
246: } /* }}} */
247:
248: static void spl_ptr_llist_copy(spl_ptr_llist *from, spl_ptr_llist *to) /* {{{ */ {
249:     spl_ptr_llist_element *current = from->head, *next;
250:     //??? spl_ptr_llist_ctor_func ctor = from->ctor;
251:     while (current) {
252:         next = current->next;
253:         //??? FIXME
254:         if (ctor) {
255:             ctor(current);
256:         }
257:         spl_ptr_llist_push(to, current->data);
258:         current = next;
259:     }
260:     if (intern->gc_data != NULL) {
261:         efree(intern->gc_data);
262:     }
263:     spl_ptr_llist_destroy(intern->llist);
264:     SPL_LLIST_CHECK_DELEAF(intern->traverse_pointer);
265:     /* }}} */
266:
267: static void spl_dlist_object_free_storage(zend_object *object) /* {{{ */ {
268:     spl_dlist_object *intern = spl_dlist_from_obj(object);
269:     zval tmp;
270:     zend_object_std_dtor(intern->std);
271:     while (intern->llist->count > 0) {
272:         spl_ptr_llist_pop(intern->llist, &tmp);
273:         spl_ptr_llist_dtor(intern->llist, &tmp);
274:     }
275:     if (intern->gc_data != NULL) {
276:         efree(intern->gc_data);
277:     }
278:     return;
279: } /* }}} */
280:
281: static zend_object *spl_dlist_object_new_ext(zend_class_entry *class_type, zval *orig, int clone_orig) /* {{{ */ {
282:     spl_dlist_object *intern;
283:     zend_class_entry *parent = class_type;
284:     int inherited = 0;
285:     intern = zend_object_alloc(sizeof(spl_dlist_object), parent);
286:     return intern;
287: } /* }}} */

```

```

377: zend_object_std_init(&intern->std, class_type);
378: object_properties_init(&intern->std, class_type);
379:
380:
381: intern->flags = 0;
382: intern->traverse_position = 0;
383:
384: if (orig) {
385:     spl_dlist_object *other = Z_SPDOLLIST_P(orig);
386:     intern->oc_get_iterator = other->oc_get_iterator;
387:
388:     if (clone_orig) {
389:         intern->llist = (spl_ptr_llist *)spl_ptr_llist_init(&other->llist->ctor, &other->llist->dtor);
390:         spl_ptr_llist_copy(&other->llist, intern->llist);
391:         intern->traverse_pointer = intern->llist->thead;
392:         SPL_LLIST_CHECK_ADDRP(intern->traverse_pointer);
393:     } else {
394:         intern->llist = other->llist;
395:         intern->traverse_pointer = intern->llist->thead;
396:         SPL_LLIST_CHECK_ADDRP(intern->traverse_pointer);
397:     }
398:
399:     intern->flags = other->flags;
400: } else {
401:     intern->llist = (spl_ptr_llist *)spl_ptr_llist_init(spl_ptr_llist_ctor_val, spl_ptr_llist_dtor);
402:     intern->traverse_pointer = intern->llist->thead;
403:     SPL_LLIST_CHECK_ADDRP(intern->traverse_pointer);
404: }
405:
406: while (parent) {
407:     if (parent == spl_oc_SplStack) {
408:         intern->flags |= (SPL_DLIST_IT_FIX | SPL_DLIST_IT_IFPO);
409:         intern->std.handlers = spl_handler_SplDoublyLinkedList;
410:     } else if (parent == spl_oc_SplQueue) {
411:         intern->flags |= (SPL_DLIST_IT_FIX);
412:         intern->std.handlers = spl_handler_SplDoublyLinkedList;
413:     }
414:
415:     if (parent == spl_oc_SplDoublyLinkedList) {
416:         intern->std.handlers = spl_handler_SplDoublyLinkedList;
417:         break;
418:     }
419:
420:     parent = parent->parent;
421:     inherited = 1;
422: }
423:
424: if (!parent) { /* this must never happen */
425:     php_error_docref(NULL, E_COMPILE_ERROR, "Internal compiler error. Class is not child of SplDoublyLinkedList");
426: }
427:
428: if (inherited) {
429:     intern->fptr_offset_get = zend_hash_str_find_ptr(class_type->function_table, "offsetget", sizeof("offsetget") - 1);
430:     if (intern->fptr_offset_get->common.scope == parent) {
431:         intern->fptr_offset_get = NULL;
432:     }
433:     intern->fptr_offset_set = zend_hash_str_find_ptr(class_type->function_table, "offsetset", sizeof("offsetset") - 1);
434:     if (intern->fptr_offset_set->common.scope == parent) {
435:         intern->fptr_offset_set = NULL;
436:     }
437:     intern->fptr_offset_has = zend_hash_str_find_ptr(class_type->function_table, "offsetexists", sizeof("offsetexists") - 1);
438:     if (intern->fptr_offset_has->common.scope == parent) {
439:         intern->fptr_offset_has = NULL;
440:     }
441:     intern->fptr_offset_del = zend_hash_str_find_ptr(class_type->function_table, "offsetunset", sizeof("offsetunset") - 1);
442:     if (intern->fptr_offset_del->common.scope == parent) {
443:         intern->fptr_offset_del = NULL;
444:     }
445:     intern->fptr_count = zend_hash_str_find_ptr(class_type->function_table, "count", sizeof("count") - 1);
446:     if (intern->fptr_count->common.scope == parent) {
447:         intern->fptr_count = NULL;
448:     }
449: }
450:
451: return &intern->std;
452: } /* }}} */
453:
454: static zend_object *spl_dlist_object_new(zend_class_entry *class_type) /* {{{ */
455: {
456:     return spl_dlist_object_new_ex(class_type, NULL, 0);
457: }
458: /* }}} */
459:
460: static zend_object *spl_dlist_object_clone(zval *zobject) /* {{{ */
461: {
462:     zend_object *old_object;
463:     zend_object *new_object;
464:
465:     old_object = Z_OBJ_P(zobject);
466:     new_object = spl_dlist_object_new_ex(old_object->ce, zobject, 1);
467:
468:     zend_objects_clone_members(new_object, old_object);
469:
470:     return new_object;
471: }
472: /* }}} */
473:
474: static int spl_dlist_object_count_elements(zval *zobject, zend_long *count) /* {{{ */
475: {
476:     spl_dlist_object *intern = Z_SPDOLLIST_P(zobject);
477:
478:     if (intern->fptr_count) {
479:         zend_call_method_with_0_params(object, intern->std.ce, intern->fptr_count, "count", &rv);
480:         if (!Z_ISNUMERIC(rv)) {
481:             *count = zval_get_long(&rv);
482:             zval_get_dtor(&rv);
483:             return SUCCESS;
484:         }
485:         *count = 0;
486:         return FAILURE;
487:     }
488:
489:     *count = spl_ptr_llist_count(intern->llist);
490:     return SUCCESS;
491: }
492: /* }}} */
493:
494: static HashTable* spl_dlist_object_get_debug_info(zval *ob, int *is_a_temp) /* {{{ */
495: {
496:     spl_dlist_object *intern = Z_SPDOLLIST_P(ob);
497:     spl_ptr_llist_element *current = intern->llist->thead;
498:     zval tmp, dlist_array;
499:     zend_string *pnstr;
500:     int i = 0;
501:     HashTable *debug_info;
502:     *is_a_temp = 1;
503:
504:     if (!(&intern->std.properties)) {
505:         rebuild_object_properties(&intern->std);
506:     }
507:
508:     debug_info = zend_new_array(1);
509:     zend_hash_copy(debug_info, intern->std.properties, (copy_ctor_func_t) zval_add_ref);
510:
511:     pnstr = spl_gen_private_prop_name(spl_oc_SplDoublyLinkedList, "Flags", sizeof("Flags")-1);
512:     ZVAL_LONG(&tmp, intern->flags);
513:     zend_hash_add(debug_info, pnstr, &tmp);
514:     zend_string_release(pnstr);
515:
516:     array_init(&dlist_array);
517:
518:     while (current) {
519:         next = current->next;
520:
521:         add_index_zval(&dlist_array, i, current->data);
522:         if (Z_REFCOUNTED(current->data)) {
523:             Z_ADDREF(current->data);
524:         }
525:         i++;
526:         current = next;
527:     }
528:
529:     pnstr = spl_gen_private_prop_name(spl_oc_SplDoublyLinkedList, "dlist", sizeof("dlist")-1);
530:     zend_hash_add(debug_info, pnstr, &dlist_array);
531:     zend_string_release(pnstr);
532:
533:     return debug_info;
534: }
535: /* }}} */
536:
537: static HashTable *spl_dlist_object_get_gc(zval *ob, zval **gc_data, int *gc_data_count) /* {{{ */
538: {
539:     spl_dlist_object *intern = Z_SPDOLLIST_P(ob);
540:
541:     spl_ptr_llist_element *current = intern->llist->thead;
542:     int i = 0;
543:
544:     if (intern->gc_data_count < intern->llist->count) {
545:         intern->gc_data_count = intern->llist->count;
546:         intern->gc_data = safe_erealloc(intern->gc_data, intern->gc_data_count, sizeof(zval), 0);
547:     }
548:
549:     while (current) {
550:         ZVAL_COPY_VALUE(intern->gc_data[i++], &current->data);
551:         current = current->next;
552:     }
553:
554:     *gc_data = intern->gc_data;
555:     *gc_data_count = i;
556:     return zend_std_get_properties(ob);
557: }
558: /* }}} */
559:
560: /* {{{ proto bool SplDoublyLinkedList::push(mixed value)
561: Push value on the SplDoublyLinkedList */
562: SPL_METHOD(SplDoublyLinkedList, push)
563: {

```

```

564:     zval *value;
565:     spl_dlist_object *intern;
566:
567:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &value) == FAILURE) {
568:         return;
569:     }
570:
571:     intern = Z_SPDOLLIST_P(getThis());
572:     spl_ptr_llist_push(intern->llist, value);
573:
574:     RETURN_TRUE;
575: }
576:
577: /* }}} */
578:
579: /* {{{ proto bool SplDoublyLinkedList::unshift(mixed value)
580: Unshift value on the SplDoublyLinkedList */
581: SPL_METHOD(SplDoublyLinkedList, unshift)
582: {
583:     zval *value;
584:     spl_dlist_object *intern;
585:
586:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &value) == FAILURE) {
587:         return;
588:     }
589:
590:     intern = Z_SPDOLLIST_P(getThis());
591:     spl_ptr_llist_unshift(intern->llist, value);
592:
593:     RETURN_TRUE;
594: }
595: /* }}} */
596:
597: /* {{{ proto mixed SplDoublyLinkedList::pop()
598: Pop an element out of the SplDoublyLinkedList */
599: SPL_METHOD(SplDoublyLinkedList, pop)
600: {
601:     spl_dlist_object *intern;
602:
603:     if (zend_parse_parameters_none() == FAILURE) {
604:         return;
605:     }
606:
607:     intern = Z_SPDOLLIST_P(getThis());
608:     spl_ptr_llist_pop(intern->llist, return_value);
609:
610:     if (Z_ISUNDEF_P(return_value)) {
611:         zend_throw_exception(spl_oc_RuntimeException, "Can't pop from an empty datastructure", 0);
612:         RETURN_NULL();
613:     }
614:
615:     /* }}} */
616:
617: /* {{{ proto mixed SplDoublyLinkedList::shift()
618: Shift an element out of the SplDoublyLinkedList */
619: SPL_METHOD(SplDoublyLinkedList, shift)
620: {
621:     spl_dlist_object *intern;
622:
623:     if (zend_parse_parameters_none() == FAILURE) {
624:         return;
625:     }
626:
627:     intern = Z_SPDOLLIST_P(getThis());
628:     spl_ptr_llist_shift(intern->llist, return_value);
629:
630:     if (Z_ISUNDEF_P(return_value)) {
631:         zend_throw_exception(spl_oc_RuntimeException, "Can't shift from an empty datastructure", 0);
632:         RETURN_NULL();
633:     }
634:
635:     /* }}} */
636:
637: /* {{{ proto mixed SplDoublyLinkedList::top()
638: Peek at the top element of the SplDoublyLinkedList */
639: SPL_METHOD(SplDoublyLinkedList, top)
640: {
641:     zval *value;
642:     spl_dlist_object *intern;
643:
644:     if (zend_parse_parameters_none() == FAILURE) {
645:         return;
646:     }
647:
648:     intern = Z_SPDOLLIST_P(getThis());
649:     value = spl_ptr_llist_last(intern->llist);
650:
651:     if (value == NULL || Z_ISUNDEF_P(value)) {
652:         zend_throw_exception(spl_oc_RuntimeException, "Can't peek at an empty datastructure", 0);
653:         return;
654:     }
655:
656:     ZVAL_DEREF(value);
657:     ZVAL_COPY(return_value, value);
658:
659:     /* }}} */
660:
661: /* {{{ proto mixed SplDoublyLinkedList::bottom()
662: Peek at the bottom element of the SplDoublyLinkedList */
663: SPL_METHOD(SplDoublyLinkedList, bottom)
664: {
665:     zval *value;
666:     spl_dlist_object *intern;
667:
668:     if (zend_parse_parameters_none() == FAILURE) {
669:         return;
670:     }
671:
672:     intern = Z_SPDOLLIST_P(getThis());
673:     value = spl_ptr_llist_first(intern->llist);
674:
675:     if (value == NULL || Z_ISUNDEF_P(value)) {
676:         zend_throw_exception(spl_oc_RuntimeException, "Can't peek at an empty datastructure", 0);
677:         return;
678:     }
679:
680:     ZVAL_DEREF(value);
681:     ZVAL_COPY(return_value, value);
682:
683:     /* }}} */
684:
685: /* {{{ proto int SplDoublyLinkedList::count()
686: Return the number of elements in the datastructure. */
687: SPL_METHOD(SplDoublyLinkedList, count)
688: {
689:     zend_long count;
690:     spl_dlist_object *intern = Z_SPDOLLIST_P(getThis());
691:
692:     if (zend_parse_parameters_none() == FAILURE) {
693:         return;
694:     }
695:
696:     count = spl_ptr_llist_count(intern->llist);
697:     RETURN_LONG(count);
698: }
699: /* }}} */
700:
701: /* {{{ proto int SplDoublyLinkedList::isEmpty()
702: Return true if the SplDoublyLinkedList is empty. */
703: SPL_METHOD(SplDoublyLinkedList, isEmpty)
704: {
705:     zend_long count;
706:
707:     if (zend_parse_parameters_none() == FAILURE) {
708:         return;
709:     }
710:
711:     spl_dlist_object_count_elements(getThis(), &count);
712:     RETURN_BOOL(count == 0);
713: }
714: /* }}} */
715:
716: /* {{{ proto int SplDoublyLinkedList::setIteratorMode(int flags)
717: Set the mode of iteration */
718: SPL_METHOD(SplDoublyLinkedList, setIteratorMode)
719: {
720:     zend_long value;
721:     spl_dlist_object *intern;
722:
723:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "l", &value) == FAILURE) {
724:         return;
725:     }
726:
727:     intern = Z_SPDOLLIST_P(getThis());
728:
729:     if (intern->flags & SPL_DLIST_IT_FIX)
730:         & (intern->flags & SPL_DLIST_IT_IFPO) |= (value & SPL_DLIST_IT_IFPO);
731:     zend_throw_exception(spl_oc_RuntimeException, "Iterators' LIFO/FIFO modes for SplStack/SplQueue objects are frozen", 0);
732:     return;
733: }
734:
735: intern->flags = (value & SPL_DLIST_IT_MASK) | (intern->flags & SPL_DLIST_IT_FIX);
736:
737: RETURN_LONG(intern->flags);
738: }
739: /* }}} */
740:
741: /* {{{ proto int SplDoublyLinkedList::getIteratorMode()
742: Return the mode of iteration */
743: SPL_METHOD(SplDoublyLinkedList, getIteratorMode)
744: {
745:     spl_dlist_object *intern;
746:
747:     if (zend_parse_parameters_none() == FAILURE) {
748:         return;
749:     }
750:
751:     intern = Z_SPDOLLIST_P(getThis());
752:

```

```

752: RETURN_LONG(intern->flags);
754: }
755: /* }}} */
756:
757: /* {{{ proto bool SplDoublyLinkedList::offsetExists(mixed index)
758: Returns whether the requested $index exists. */
759: SPL_METHOD(SplDoublyLinkedList, offsetExists)
760: {
761:     zval *zindex;
762:     spl_dlist_object *intern;
763:     zend_long index;
764:
765:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &zindex) == FAILURE) {
766:         return;
767:     }
768:
769:     intern = Z_SPDLIST_P(getThis());
770:     index = spl_offset_convert_to_long(index);
771:
772:     RETURN_BOOL(index >= 0 && index < intern->vlist->count);
773: } /* }}} */
774:
775: /* {{{ proto mixed SplDoublyLinkedList::offsetGet(mixed index)
776: Returns the value at the specified $index. */
777: SPL_METHOD(SplDoublyLinkedList, offsetGet)
778: {
779:     zval *zindex;
780:     zend_long index;
781:     spl_dlist_object *intern;
782:     spl_ptr_list_element *element;
783:
784:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &zindex) == FAILURE) {
785:         return;
786:     }
787:
788:     intern = Z_SPDLIST_P(getThis());
789:     index = spl_offset_convert_to_long(index);
790:
791:     if (index < 0 || index > intern->vlist->count) {
792:         zend_throw_exception(spl_ce_OutOfRangeException, "Offset invalid or out of range", 0);
793:         return;
794:     }
795:
796:     element = spl_ptr_list_offset(intern->vlist, index, intern->flags & SPL_DLIST_IT_LIFO);
797:
798:     if (element != NULL) {
799:         zval *value = element->data;
800:
801:         ZVAL_DEREF(value);
802:         ZVAL_COPY(return_value, value);
803:     } else {
804:         zend_throw_exception(spl_ce_OutOfRangeException, "Offset invalid", 0);
805:     }
806: } /* }}} */
807:
808: /* {{{ proto void SplDoublyLinkedList::offsetSet(mixed index, mixed value)
809: Sets the value at the specified $index to $value. */
810: SPL_METHOD(SplDoublyLinkedList, offsetSet)
811: {
812:     zval *zindex, *value;
813:     spl_dlist_object *intern;
814:
815:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "an", &zindex, &value) == FAILURE) {
816:         return;
817:     }
818:
819:     intern = Z_SPDLIST_P(getThis());
820:
821:     if (Z_TYPE_P(zindex) == IS_NULL) {
822:         /* $obj[] = ... */
823:         spl_ptr_list_push(intern->vlist, value);
824:     } else {
825:         /* $obj[$foo] = ... */
826:         zend_long index;
827:         spl_ptr_list_element *element;
828:
829:         index = spl_offset_convert_to_long(index);
830:
831:         if (index < 0 || index > intern->vlist->count) {
832:             zend_throw_exception(spl_ce_OutOfRangeException, "Offset invalid or out of range", 0);
833:             return;
834:         }
835:
836:         element = spl_ptr_list_offset(intern->vlist, index, intern->flags & SPL_DLIST_IT_LIFO);
837:
838:         if (element != NULL) {
839:             /* call dtor on the old element as in spl_ptr_list_pop */
840:             if (intern->vlist->dtor) {
841:                 intern->vlist->dtor(element);
842:             }
843:
844:             /* the element is replaced, deref the old one as in
845:              * SplDoublyLinkedList::pop() */
846:             zval_ptr_dtor(&element->data);
847:             ZVAL_COPY_VALUE(element->data, value);
848:
849:             /* new element, call ctor as in spl_ptr_list_push */
850:             if (intern->vlist->ctor) {
851:                 intern->vlist->ctor(element);
852:             }
853:         } else {
854:             zval_ptr_dtor(value);
855:             zend_throw_exception(spl_ce_OutOfRangeException, "Offset invalid", 0);
856:             return;
857:         }
858:     }
859: } /* }}} */
860:
861: /* {{{ proto void SplDoublyLinkedList::offsetUnset(mixed index)
862: Unsets the value at the specified $index. */
863: SPL_METHOD(SplDoublyLinkedList, offsetUnset)
864: {
865:     zval *zindex;
866:     zend_long index;
867:     spl_dlist_object *intern;
868:     spl_ptr_list_element *element;
869:     spl_ptr_list *vlist;
870:
871:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &zindex) == FAILURE) {
872:         return;
873:     }
874:
875:     intern = Z_SPDLIST_P(getThis());
876:     index = spl_offset_convert_to_long(index);
877:     vlist = intern->vlist;
878:
879:     if (index < 0 || index > intern->vlist->count) {
880:         zend_throw_exception(spl_ce_OutOfRangeException, "Offset out of range", 0);
881:         return;
882:     }
883:
884:     element = spl_ptr_list_offset(intern->vlist, index, intern->flags & SPL_DLIST_IT_LIFO);
885:
886:     if (element != NULL) {
887:         /* connect the neighbors */
888:         if (element->prev) {
889:             element->prev->next = element->next;
890:         }
891:
892:         if (element->next) {
893:             element->next->prev = element->prev;
894:         }
895:
896:         /* take care of head/tail */
897:         if (element == vlist->head) {
898:             vlist->head = element->next;
899:         }
900:
901:         if (element == vlist->tail) {
902:             vlist->tail = element->prev;
903:         }
904:
905:         /* finally, delete the element */
906:         vlist->count--;
907:
908:         if (vlist->dtor) {
909:             vlist->dtor(element);
910:         }
911:
912:         if (intern->traverse_pointer == element) {
913:             SPL_LIST_DELREF(element);
914:             intern->traverse_pointer = NULL;
915:         }
916:         zval_ptr_dtor(&element->data);
917:         ZVAL_UNREF(element->data);
918:
919:         SPL_LIST_DELREF(element);
920:     } else {
921:         zend_throw_exception(spl_ce_OutOfRangeException, "Offset invalid", 0);
922:         return;
923:     }
924: } /* }}} */
925:
926: static void spl_dlist_it_dtor(zend_object_iterator *iter) /* {{{ */
927: {
928:     spl_dlist_it *iterator = (spl_dlist_it *)iter;
929:
930:     SPL_LIST_CHECK_DELREF(iterator->traverse_pointer);
931:
932:     zend_user_it_invalidate_current(iterator);
933:     zval_ptr_dtor(&iterator->intern.it.data);
934: }
935: /* }}} */
936:
937: static void spl_dlist_it_helper_rewind(spl_ptr_list_element **traverse_pointer_ptr, int *traverse_position_ptr, spl_ptr_list *vlist, int flags) /*

```

```

940:
941: if (flags & SPL_DLIST_IT_LIFO) {
942:     *traverse_position_ptr = vlist->count-1;
943:     *traverse_pointer_ptr = vlist->tail;
944: } else {
945:     *traverse_position_ptr = 0;
946:     *traverse_pointer_ptr = vlist->head;
947: }
948:
949: SPL_LIST_CHECK_ADREF(*traverse_pointer_ptr);
950: }
951: /* }}} */
952:
953: static void spl_dlist_it_helper_move_forward(spl_ptr_list_element **traverse_pointer_ptr, int *traverse_position_ptr, spl_ptr_list *vlist, int flags)
954: /* {{{ */
955: {
956:     if (*traverse_pointer_ptr) {
957:         spl_ptr_list_element *old = *traverse_pointer_ptr;
958:
959:         if (flags & SPL_DLIST_IT_LIFO) {
960:             *traverse_position_ptr = old->prev;
961:             (*traverse_position_ptr)--;
962:         }
963:
964:         if (flags & SPL_DLIST_IT_DELETE) {
965:             zval prev;
966:             spl_ptr_list_pop(vlist, &prev);
967:
968:             zval_ptr_dtor(&prev);
969:         } else {
970:             *traverse_position_ptr = old->next;
971:
972:             if (flags & SPL_DLIST_IT_DELETE) {
973:                 zval prev;
974:                 spl_ptr_list_shift(vlist, &prev);
975:
976:                 zval_ptr_dtor(&prev);
977:             }
978:             (*traverse_position_ptr)++;
979:         }
980:     }
981:
982:     SPL_LIST_CHECK_DELREF(old);
983:     SPL_LIST_CHECK_ADREF(*traverse_pointer_ptr);
984: }
985: /* }}} */
986:
987: static void spl_dlist_it_rewind(zend_object_iterator *iter) /* {{{ */
988: {
989:     spl_dlist_it *iterator = (spl_dlist_it *)iter;
990:     spl_dlist_object *object = Z_SPDLIST_P(iterator->data);
991:     spl_ptr_list *vlist = object->vlist;
992:
993:     spl_dlist_it_helper_rewind(&iterator->traverse_pointer, &iterator->traverse_position, vlist, object->flags);
994: }
995: /* }}} */
996:
997: static int spl_dlist_it_valid(zend_object_iterator *iter) /* {{{ */
998: {
999:     spl_dlist_it *iterator = (spl_dlist_it *)iter;
1000:     spl_ptr_list_element *element = iterator->traverse_pointer;
1001:
1002:     return (element != NULL ? SUCCESS : FAILURE);
1003: }
1004: /* }}} */
1005:
1006: static zval *spl_dlist_it_get_current_data(zend_object_iterator *iter) /* {{{ */
1007: {
1008:     spl_dlist_it *iterator = (spl_dlist_it *)iter;
1009:     spl_ptr_list_element *element = iterator->traverse_pointer;
1010:
1011:     if (element == NULL || Z_UNDEF(element->data)) {
1012:         return NULL;
1013:     }
1014:
1015:     return element->data;
1016: }
1017: /* }}} */
1018:
1019: static void spl_dlist_it_get_current_key(zend_object_iterator *iter, zval *key) /* {{{ */
1020: {
1021:     spl_dlist_it *iterator = (spl_dlist_it *)iter;
1022:
1023:     ZVAL_LONG(key, iterator->traverse_position);
1024: }
1025: /* }}} */
1026:
1027: static void spl_dlist_it_move_forward(zend_object_iterator *iter) /* {{{ */
1028: {
1029:     spl_dlist_it *iterator = (spl_dlist_it *)iter;
1030:     spl_dlist_object *object = Z_SPDLIST_P(iterator->data);
1031:
1032:     zend_user_it_invalidate_current(iter);
1033:
1034:     spl_dlist_it_helper_move_forward(&iterator->traverse_pointer, &iterator->traverse_position, object->vlist, object->flags);
1035: }
1036: /* }}} */
1037:
1038: /* {{{ proto int SplDoublyLinkedList::key()
1039: Return current array key */
1040: SPL_METHOD(SplDoublyLinkedList, key)
1041: {
1042:     spl_dlist_object *intern = Z_SPDLIST_P(getThis());
1043:
1044:     if (zend_parse_parameters_none() == FAILURE) {
1045:         return;
1046:     }
1047:
1048:     RETURN_LONG(intern->traverse_position);
1049: }
1050: /* }}} */
1051:
1052: /* {{{ proto void SplDoublyLinkedList::prev()
1053: Move to next entry */
1054: SPL_METHOD(SplDoublyLinkedList, prev)
1055: {
1056:     spl_dlist_object *intern = Z_SPDLIST_P(getThis());
1057:
1058:     if (zend_parse_parameters_none() == FAILURE) {
1059:         return;
1060:     }
1061:
1062:     spl_dlist_it_helper_move_forward(&intern->traverse_pointer, &intern->traverse_position, intern->vlist, intern->flags & SPL_DLIST_IT_LIFO);
1063: }
1064: /* }}} */
1065:
1066: /* {{{ proto void SplDoublyLinkedList::next()
1067: Move to next entry */
1068: SPL_METHOD(SplDoublyLinkedList, next)
1069: {
1070:     spl_dlist_object *intern = Z_SPDLIST_P(getThis());
1071:
1072:     if (zend_parse_parameters_none() == FAILURE) {
1073:         return;
1074:     }
1075:
1076:     spl_dlist_it_helper_move_forward(&intern->traverse_pointer, &intern->traverse_position, intern->vlist, intern->flags);
1077: }
1078: /* }}} */
1079:
1080: /* {{{ proto bool SplDoublyLinkedList::valid()
1081: Check whether the datastructure contains more entries */
1082: SPL_METHOD(SplDoublyLinkedList, valid)
1083: {
1084:     spl_dlist_object *intern = Z_SPDLIST_P(getThis());
1085:
1086:     if (zend_parse_parameters_none() == FAILURE) {
1087:         return;
1088:     }
1089:
1090:     RETURN_BOOL(intern->traverse_pointer != NULL);
1091: }
1092: /* }}} */
1093:
1094: /* {{{ proto void SplDoublyLinkedList::rewind()
1095: Rewind the datastructure back to the start */
1096: SPL_METHOD(SplDoublyLinkedList, rewind)
1097: {
1098:     spl_dlist_object *intern = Z_SPDLIST_P(getThis());
1099:
1100:     if (zend_parse_parameters_none() == FAILURE) {
1101:         return;
1102:     }
1103:
1104:     spl_dlist_it_helper_rewind(&intern->traverse_pointer, &intern->traverse_position, intern->vlist, intern->flags);
1105: }
1106: /* }}} */
1107:
1108: /* {{{ proto mixed SplDoublyLinkedList::current()
1109: Return current datastructure entry */
1110: SPL_METHOD(SplDoublyLinkedList, current)
1111: {
1112:     spl_dlist_object *intern = Z_SPDLIST_P(getThis());
1113:     spl_ptr_list_element *element = intern->traverse_pointer;
1114:
1115:     if (zend_parse_parameters_none() == FAILURE) {
1116:         return;
1117:     }
1118:
1119:     if (element == NULL || Z_UNDEF(element->data)) {
1120:         RETURN_NULL();
1121:     } else {
1122:         zval *value = element->data;
1123:
1124:         ZVAL_DEREF(value);
1125:         ZVAL_COPY(return_value, value);
1126:     }

```

[illegible]

```

3118:     EVAL_UNDEF(iterator->intern,value);
3119: }
3120: SPL_LIST_CHECK_ADDRESS(iterator->traverse_pointer);
3121:
3122: // }}}
3123:
3124: /* Function/Class/Method definitions */
3125: ZEND_BEGIN_ARG_INFO(arginfo_dlist_setteratormode, 0)
3126: {
3127:     ZEND_ARG_INFO(0, flags)
3128: }
3129: ZEND_BEGIN_ARG_INFO(arginfo_dlist_push, 0)
3130: {
3131:     ZEND_ARG_INFO(0, value)
3132: }
3133: ZEND_END_ARG_INFO()
3134:
3135: ZEND_BEGIN_ARG_INFO_EX(arginfo_dlist_offsetset, 0, 0, 1)
3136: {
3137:     ZEND_ARG_INFO(0, index)
3138: }
3139: ZEND_END_ARG_INFO()
3140:
3141: ZEND_BEGIN_ARG_INFO(arginfo_dlist_void, 0)
3142: {
3143:     ZEND_ARG_INFO(0, newval)
3144: }
3145: ZEND_END_ARG_INFO()
3146:
3147: SPL_MK_FUNC(arginfo_dlist_void, 0)
3148:
3149: ZEND_BEGIN_ARG_INFO(arginfo_dlist_serialized, 0)
3150: {
3151:     ZEND_ARG_INFO(0, serialized)
3152: }
3153:
3154: static const zend_function_entry spl_funcs_sp1queue[] = {
3155:     SPL_MK_FUNC(queue, SPL_DLIST_LINKEDLIST, push, arginfo_dlist_push, ZEND_ACC_PUBLIC)
3156:     SPL_MK_FUNC(dequeue, SPL_DLIST_LINKEDLIST, shift, arginfo_dlist_void, ZEND_ACC_PUBLIC)
3157:     PHP_FE_END
3158: };
3159:
3160: static const zend_function_entry spl_funcs_sp1doublylinkedlist[] = {
3161:     SPL_MK_FUNC(sp1doublylinkedlist, pop, arginfo_dlist_void, ZEND_ACC_PUBLIC)
3162:     SPL_MK_FUNC(sp1doublylinkedlist, shift, arginfo_dlist_void, ZEND_ACC_PUBLIC)
3163:     SPL_MK_FUNC(sp1doublylinkedlist, push, arginfo_dlist_push, ZEND_ACC_PUBLIC)
3164:     SPL_MK_FUNC(sp1doublylinkedlist, unshift, arginfo_dlist_push, ZEND_ACC_PUBLIC)
3165:     SPL_MK_FUNC(sp1doublylinkedlist, top, arginfo_dlist_void, ZEND_ACC_PUBLIC)
3166:     SPL_MK_FUNC(sp1doublylinkedlist, bottom, arginfo_dlist_void, ZEND_ACC_PUBLIC)
3167:     SPL_MK_FUNC(sp1doublylinkedlist, isempty, arginfo_dlist_void, ZEND_ACC_PUBLIC)
3168:     SPL_MK_FUNC(sp1doublylinkedlist, setIteratorMode, arginfo_dlist_setteratormode, ZEND_ACC_PUBLIC)
3169:     SPL_MK_FUNC(sp1doublylinkedlist, getIteratorMode, arginfo_dlist_void, ZEND_ACC_PUBLIC)
3170:     /* Countable */
3171:     SPL_MK_FUNC(sp1doublylinkedlist, count, arginfo_dlist_void, ZEND_ACC_PUBLIC)
3172:     /* ArrayAccess */
3173:     SPL_MK_FUNC(sp1doublylinkedlist, offsetExists, arginfo_dlist_offsetset, ZEND_ACC_PUBLIC)
3174:     SPL_MK_FUNC(sp1doublylinkedlist, offsetGet, arginfo_dlist_offsetset, ZEND_ACC_PUBLIC)
3175:     SPL_MK_FUNC(sp1doublylinkedlist, offsetSet, arginfo_dlist_offsetset, ZEND_ACC_PUBLIC)
3176:     SPL_MK_FUNC(sp1doublylinkedlist, offsetUnset, arginfo_dlist_offsetset, ZEND_ACC_PUBLIC)
3177:
3178:     SPL_MK_FUNC(sp1doublylinkedlist, add, arginfo_dlist_offsetset, ZEND_ACC_PUBLIC)
3179:
3180:     /* Iterator */
3181:     SPL_MK_FUNC(sp1doublylinkedlist, rewind, arginfo_dlist_void, ZEND_ACC_PUBLIC)
3182:     SPL_MK_FUNC(sp1doublylinkedlist, current, arginfo_dlist_void, ZEND_ACC_PUBLIC)
3183:     SPL_MK_FUNC(sp1doublylinkedlist, key, arginfo_dlist_void, ZEND_ACC_PUBLIC)
3184:     SPL_MK_FUNC(sp1doublylinkedlist, next, arginfo_dlist_void, ZEND_ACC_PUBLIC)
3185:     SPL_MK_FUNC(sp1doublylinkedlist, prev, arginfo_dlist_void, ZEND_ACC_PUBLIC)
3186:     SPL_MK_FUNC(sp1doublylinkedlist, valid, arginfo_dlist_void, ZEND_ACC_PUBLIC)
3187:     /* Serializable */
3188:     SPL_MK_FUNC(sp1doublylinkedlist, unserialize, arginfo_dlist_serialized, ZEND_ACC_PUBLIC)
3189:     SPL_MK_FUNC(sp1doublylinkedlist, serialize, arginfo_dlist_void, ZEND_ACC_PUBLIC)
3190:     PHP_FE_END
3191: };
3192:
3193: /* }}}
3194:
3195: PHP_MINIT_FUNCTION(spl_dlist) { /* {{{ */
3196:     {
3197:         REGISTER_SPL_STD_CLASS_EX(spldoublylinkedlist, spl_dlist_object_new, spl_funcs_sp1doublylinkedlist);
3198:         memcpy(spl_handler_sp1doublylinkedlist, zend_get_std_obj_handlers(), sizeof(zend_obj_handlers));
3199:
3200:         spl_handler_sp1doublylinkedlist->offset = XOFFSET(spl_dlist_object, spl);
3201:         spl_handler_sp1doublylinkedlist->clone_obj = spl_dlist_object_clone;
3202:         spl_handler_sp1doublylinkedlist->count_elements = spl_dlist_object_count_elements;
3203:         spl_handler_sp1doublylinkedlist->get_debug_info = spl_dlist_object_get_debug_info;
3204:         spl_handler_sp1doublylinkedlist->get_gc = spl_dlist_object_get_gc;
3205:         spl_handler_sp1doublylinkedlist->for_obj = zend_obj_dlist_destroy_obj;
3206:         spl_handler_sp1doublylinkedlist->free_obj = spl_dlist_object_free_storage;
3207:     }
3208:
3209:     REGISTER_SPL_CLASS_CONST_LONG(spldoublylinkedlist, "IT_MODE_LIST", SPL_DLIST_IT_LIST);
3210:     REGISTER_SPL_CLASS_CONST_LONG(spldoublylinkedlist, "IT_MODE_FIFO", 0);
3211:     REGISTER_SPL_CLASS_CONST_LONG(spldoublylinkedlist, "IT_MODE_DEQUEUE", SPL_DLIST_IT_DEQUEUE);
3212:     REGISTER_SPL_CLASS_CONST_LONG(spldoublylinkedlist, "IT_MODE_INSERT", 0);
3213:
3214:     REGISTER_SPL_IMPLEMENTATIONS(spldoublylinkedlist, Iterator);
3215:     REGISTER_SPL_IMPLEMENTATIONS(spldoublylinkedlist, Countable);
3216:     REGISTER_SPL_IMPLEMENTATIONS(spldoublylinkedlist, ArrayAccess);
3217:     REGISTER_SPL_IMPLEMENTATIONS(spldoublylinkedlist, Serializable);
3218: }
3219:
3220: spl_obj_sp1doublylinkedlist->get_iterator = spl_dlist_get_iterator;
3221:
3222:
3223: REGISTER_SPL_FUNC_CLASS_EX(splqueue, spldoublylinkedlist, spl_dlist_object_new, spl_funcs_sp1queue);
3224: REGISTER_SPL_FUNC_CLASS_EX(splstack, spldoublylinkedlist, spl_dlist_object_new, NULL);
3225:
3226: spl_obj_sp1queue->get_iterator = spl_dlist_get_iterator;
3227: spl_obj_sp1stack->get_iterator = spl_dlist_get_iterator;
3228:
3229: return SUCCESS;
3230: }
3231:
3232: /* }}}
3233:
3234: /* Local variables:
3235:  * tab-width: 4
3236:  * c-basic-offset: 4
3237:  * End:
3238:  * vim600: fdm=marker
3239:  * vim: noet sw=4 ts=4
3240:  */

```

```
1: /*
2:  * -----
3:  * | PHP Version 7 |
4:  * -----
5:  * | Copyright (c) 1997-2018 The PHP Group |
6:  * -----
7:  * | This source file is subject to version 3.01 of the PHP license, |
8:  * | that is bundled with this package in the file LICENSE, and is |
9:  * | available through the world-wide-web at the following url: |
10:  * | http://www.php.net/license/3.01.txt |
11:  * | If you did not receive a copy of the PHP license and are unable to |
12:  * | obtain it through the world-wide-web, please send a note to |
13:  * | license@php.net so we can mail you a copy immediately. |
14:  * -----
15:  * | Authors: Marcus Boerger <helly@php.net> |
16:  * -----
17:  */
18:
19: #ifdef HAVE_CONFIG_H
20: # include "config.h"
21: #endif
22:
23: #include "php.h"
24: #include "php_ini.h"
25: #include "ext/standard/info.h"
26: #include "zend_interfaces.h"
27:
28: #include "spl_spl.h"
29: #include "spl_functions.h"
30: #include "spl_engine.h"
31:
32: #include "spl_array.h"
33:
34: /* {{{ spl_initialize */
35: PHPAPI void spl_initialize(zend_class_entry *pce, zval *object)
36: {
37:     object_init_ex(object, pce);
38: }
39: /* }}} */
40:
41: PHPAPI zend_long spl_offset_convert_to_long(zval *offset) /* {{{ */
42: {
43:     zend_ulong idx;
44:
45:     try_again:
46:     switch (Z_TYPE_P(offset)) {
47:         case IS_STRING:
48:             if (ZEND_HANDLE_NUMERIC(Z_STR_P(offset), idx)) {
49:                 return idx;
50:             }
51:             break;
52:         case IS_DOUBLE:
53:             return (zend_long)Z_DVAL_P(offset);
54:         case IS_LONG:
55:             return Z_LVAL_P(offset);
56:         case IS_FALSE:
57:             return 0;
58:         case IS_TRUE:
59:             return 1;
60:         case IS_REFERENCE:
61:             offset = Z_REFVAL_P(offset);
62:             goto try_again;
63:         case IS_RESOURCE:
64:             return Z_RES_HANDLE_P(offset);
65:     }
66:     return -1;
67: }
68: /* }}} */
69:
70: /*
71:  * Local Variables:
72:  * tab-width: 4
73:  * c-basic-offset: 4
74:  * End:
75:  * vim600: fdm=marker
76:  * vim: noet sw=4 ts=4
77:  */
```

```

1: /*
2:  *
3:  * PHP Version 7
4:  *
5:  * Copyright (c) 1997-2018 The PHP Group
6:  *
7:  * This source file is subject to version 3.01 of the PHP license,
8:  * that is bundled with this package in the file LICENSE, and is
9:  * available through the world-wide-web at the following url:
10:  * http://www.php.net/license/3.01.txt
11:  * If you did not receive a copy of the PHP license and are unable to
12:  * obtain it through the world-wide-web, please send a note to
13:  * license@php.net so we can mail you a copy immediately.
14:  *
15:  * Author: Marcus Boerger <chelly@php.net>
16:  */
17:
18:
19: /* $Id$ */
20:
21: #ifndef HAVE_CONFIG_H
22: #include "config.h"
23: #endif
24:
25: #include "php.h"
26: #include "php_ini.h"
27: #include "ext/standard/info.h"
28: #include "ext/standard/file.h"
29: #include "ext/standard/php_string.h"
30: #include "zend_compile.h"
31: #include "zend_exceptions.h"
32: #include "zend_interfaces.h"
33:
34: #include "php_spl.h"
35: #include "spl_functions.h"
36: #include "spl_engine.h"
37: #include "spl_iterators.h"
38: #include "spl_directory.h"
39: #include "spl_exceptions.h"
40:
41: #include "php.h"
42: #include "fopen_wrappers.h"
43:
44: #include "ext/standard/basic_functions.h"
45: #include "ext/standard/php_filestat.h"
46:
47: #define SPL_HAS_FLAG(flags, test_flag) ((flags & test_flag) ? 1 : 0)
48:
49: /* declare the class handlers */
50: static zend_object_handlers spl_filesystem_object_handlers;
51: /* Includes handler to validate object state when retrieving methods */
52: static zend_object_handlers spl_filesystem_object_check_handlers;
53:
54: /* declare the class entry */
55: PHPAPI zend_class_entry *spl_ce_SplFileInfo;
56: PHPAPI zend_class_entry *spl_ce_DirectoryIterator;
57: PHPAPI zend_class_entry *spl_ce_FilesystemIterator;
58: PHPAPI zend_class_entry *spl_ce_CurdirIterator;
59: PHPAPI zend_class_entry *spl_ce_GlobIterator;
60: PHPAPI zend_class_entry *spl_ce_SplFileObject;
61: PHPAPI zend_class_entry *spl_ce_SplTempFileObject;
62:
63: static void spl_filesystem_file_free_line(spl_filesystem_object *intern) /* {{{ */
64: {
65:     if (intern->u.file.current_line) {
66:         efree(intern->u.file.current_line);
67:         intern->u.file.current_line = NULL;
68:     }
69:     if (IS_INDEXED(intern->u.file.current_rval)) {
70:         eval_ptr_dtor(intern->u.file.current_rval);
71:         ZVAL_UNDEF(intern->u.file.current_rval);
72:     }
73: } /* }}} */
74:
75: static void spl_filesystem_object_destroy_object(spl_filesystem_object *intern) /* {{{ */
76: {
77:     spl_filesystem_object *intern = spl_filesystem_from_obj(object);
78:
79:     zend_object_dtor(object);
80:
81:     switch (intern->type) {
82:         case SPL_FS_DIR:
83:             if (intern->u.dir.dirp) {
84:                 php_stream_close(intern->u.dir.dirp);
85:                 intern->u.dir.dirp = NULL;
86:             }
87:             break;
88:         case SPL_FS_FILE:
89:             if (intern->u.file.stream) {
90:                 /*
91:                  * If (intern->u.file.zcontext) {
92:                  *     zend_list_delref(Z_RES_VAL_P(intern->zcontext));
93:                  * }
94:                  */
95:                 if ((intern->u.file.stream->is_persistent) &
96:                     !php_stream_close(intern->u.file.stream)) {
97:                     /*
98:                      * else {
99:                      *     php_stream_close(intern->u.file.stream);
100:                      * }
101:                     */
102:                     break;
103:                 default:
104:                     break;
105:             } /* }}} */
106:
107: static void spl_filesystem_object_free_storage(zend_object *object) /* {{{ */
108: {
109:     spl_filesystem_object *intern = spl_filesystem_from_obj(object);
110:
111:     if (intern->both_handler && intern->both_handler->dtor) {
112:         intern->both_handler->dtor(intern);
113:     }
114:
115:     zend_object_std_dtor(intern->std);
116:
117:     if (intern->u.path) {
118:         efree(intern->u.path);
119:     }
120:     if (intern->u.file_name) {
121:         efree(intern->u.file_name);
122:     }
123:     switch (intern->type) {
124:         case SPL_FS_DIR:
125:             break;
126:         case SPL_FS_FILE:
127:             if (intern->u.dir.sub_path) {
128:                 efree(intern->u.dir.sub_path);
129:             }
130:             break;
131:         case SPL_FS_FILE:
132:             if (intern->u.file.stream) {
133:                 if (intern->u.file.open_mode) {
134:                     efree(intern->u.file.open_mode);
135:                 }
136:                 if (intern->u.orig_path) {
137:                     efree(intern->u.orig_path);
138:                 }
139:             }
140:             break;
141:     }
142: } /* }}} */
143:
144: /* {{{ spl_ce_dir_object_new */
145: /* creates the object by
146:  * - allocating memory
147:  * - initializing the object members
148:  * - storing the object
149:  * - setting its handlers
150:  *
151:  * called from
152:  * - clone
153:  * - new
154:  */
155:
156: static zend_object *spl_filesystem_object_new_ex(zend_class_entry *class_type)
157: {
158:     spl_filesystem_object *intern;
159:
160:     intern = zend_object_alloc(sizeof(spl_filesystem_object), class_type);
161:     /* intern->type = SPL_FS_DIR; done by set 0 */
162:     intern->file_class = spl_ce_SplFileObject;
163:     intern->info_class = spl_ce_SplFileInfo;
164:
165:     zend_object_std_init(intern->std, class_type);
166:     object_properties_init(intern->std, class_type);
167:     intern->std.handlers = spl_filesystem_object_handlers;
168:
169:     return intern->std;
170: } /* }}} */
171:
172:
173: /* {{{ spl_filesystem_object_new */
174: /* See spl_filesystem_object_new_ex */
175: static zend_object *spl_filesystem_object_new(zend_class_entry *class_type)
176: {
177:     return spl_filesystem_object_new_ex(class_type);
178: }
179:
180: /* {{{ spl_filesystem_object_new_check */
181: static zend_object *spl_filesystem_object_new_check(zend_class_entry *class_type)
182: {
183:     spl_filesystem_object *ret = spl_filesystem_from_obj(spl_filesystem_object_new_ex(class_type));
184:     ret->std.handlers = spl_filesystem_object_check_handlers;
185:     return ret->std;
186: }
187:
188: /* }}} */
189:
190: PHPAPI char *spl_filesystem_object_get_path(spl_filesystem_object *intern, size_t *len) /* {{{ */
191: {
192:     #ifndef HAVE_CLOSE
193:     if (intern->type == SPL_FS_DIR) {
194:         if (php_stream_is(intern->u.dir.dirp, &php_glob_stream_ops)) {
195:             return php_glob_stream_get_path(intern->u.dir.dirp, 0, len);
196:         }
197:     }
198:     #endif
199:     if (len) {
200:         *len = intern->u.path_len;
201:     }
202:     return intern->u.path;
203: } /* }}} */
204:
205: static inline void spl_filesystem_object_get_file_name(spl_filesystem_object *intern) /* {{{ */
206: {
207:     char slash = SPL_HAS_FLAG(intern->flags, SPL_FILE_DIR_UNIPATHS) ? '/' : DEFAULT_SLASH;
208:
209:     switch (intern->type) {
210:         case SPL_FS_DIR:
211:             if (intern->u.file_name) {
212:                 php_error_docref(NULL, E_ERROR, "Object not initialized");
213:             }
214:             break;
215:         case SPL_FS_FILE:
216:             if (intern->u.file_name) {
217:                 efree(intern->u.file_name);
218:             }
219:             intern->u.file_name_len = spprintf(&intern->u.file_name, 0, "%s%s",
220:                 spl_filesystem_object_get_path(intern, NULL),
221:                 slash, intern->u.dir.entry_d_name);
222:             break;
223:     }
224: } /* }}} */
225:
226: static int spl_filesystem_dir_read(spl_filesystem_object *intern) /* {{{ */
227: {
228:     if (intern->u.dir.dirp || !php_stream_readdir(intern->u.dir.dirp, &intern->u.dir.entry)) {
229:         intern->u.dir.entry_d_name[0] = '\0';
230:         return 0;
231:     } else {
232:         return 1;
233:     }
234: }
235:
236: /* }}} */
237:
238: #define IS_SLASH_AT(pos) (IS_SLASH(pos[pos]))
239:
240: static inline int spl_filesystem_is_dot(const char *d_name) /* {{{ */
241: {
242:     return !strcmp(d_name, ".") || !strcmp(d_name, "..");
243: }
244: /* }}} */
245:
246: /* {{{ spl_filesystem_dir_open */
247: /* open a directory resource */
248: static void spl_filesystem_dir_open(spl_filesystem_object *intern, char *path)
249: {
250:     int skip_dots = SPL_HAS_FLAG(intern->flags, SPL_FILE_DIR_SKIPDOTS);
251:
252:     intern->type = SPL_FS_DIR;
253:     intern->u.path_len = strlen(path);
254:     intern->u.dir.dirp = php_stream_opendir(path, REPORT_ERRORS, PG(default_context));
255:     if (intern->u.path_len > 1 && IS_SLASH_AT(path, intern->u.path_len-1)) {
256:         intern->u.path = estrndup(path, intern->u.path_len);
257:     } else {
258:         intern->u.path = estrndup(path, intern->u.path_len);
259:     }
260:     intern->u.dir.index = 0;
261:
262:     if (EG(exception) || intern->u.dir.dirp == NULL) {
263:         intern->u.dir.entry_d_name[0] = '\0';
264:         if (EG(exception)) {
265:             /* open failed w/out notice (turned to exception due to E_THROW) */
266:             zend_throw_exception(spl_ce_RuntimeException, 0, "Failed to open directory \"%s\".", path);
267:         } else {
268:             /*
269:              * do {
270:              *     spl_filesystem_dir_read(intern);
271:              *     while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry_d_name))
272:              *         continue;
273:              * } while (1);
274:              */
275:             /* }}} */
276:             /*
277:              * static int spl_filesystem_file_open(spl_filesystem_object *intern, int use_include_path, int silent) /* {{{ */
278:              * {
279:              *     zend_tmp;
280:              *
281:              *     intern->type = SPL_FS_FILE;
282:              *
283:              *     php_stat(intern->u.file_name, intern->u.file_name_len, FS_IS_DIR, &tmp);
284:              *     if (IS_TYPE(tmp) == IS_THROW) {
285:              *         intern->u.file.open_mode = NULL;
286:              *         intern->u.file_name = NULL;
287:              *         zend_throw_exception(spl_ce_LogicException, 0, "Cannot use SplFileObject with directories");
288:              *         return FAILURE;
289:              *     }
290:              *
291:              *     intern->u.file.context = php_stream_context_from_rval(intern->u.file.zcontext, 0);
292:              *     intern->u.file.stream = php_stream_open_wrapper_ex(intern->u.file_name, intern->u.file.open_mode, (use_include_path ? USE_PATH : 0) | REPORT_ERRORS, MU
293:              *         L, intern->u.file.context);
294:              *
295:              *     if (intern->u.file_name_len != !intern->u.file.stream) {
296:              *         if (EG(exception)) {
297:              *             zend_throw_exception(spl_ce_RuntimeException, 0, "Cannot open file \"%s\", intern->u.file_name_len ? intern->u.file_name : "");
298:              *         }
299:              *         intern->u.file_name = NULL; /* until here it is not a copy */
300:              *         intern->u.file.open_mode = NULL;
301:              *         return FAILURE;
302:              *     }
303:              *
304:              *     /*
305:              *     if (intern->u.file.zcontext) {
306:              *         //zend_list_addref(Z_RES_VAL(intern->u.file.zcontext));
307:              *         Z_ADDREF_P(intern->u.file.zcontext);
308:              *     }
309:              *     */
310:              *
311:              *     if (intern->u.file_name_len > 1 && IS_SLASH_AT(intern->u.file_name, intern->u.file_name_len-1)) {
312:              *         intern->u.file_name_len--;
313:              *     }
314:              *
315:              *     intern->u.orig_path = estrndup(intern->u.file.stream->orig_path, strlen(intern->u.file.stream->orig_path));
316:              *
317:              *     intern->u.file_name = estrndup(intern->u.file_name, intern->u.file_name_len);
318:              *     intern->u.file.open_mode = estrndup(intern->u.file.open_mode, intern->u.file.open_mode_len);
319:              *
320:              *     /* avoid reference counting in debug mode, thus do it manually */
321:              *     ZVAL_RES(&intern->u.file.resource, intern->u.file.stream->res);
322:              *     /*!!! TODO: maybe bug?
323:              *     Z_SET_REFCOUNT(intern->u.file.resource, 1);
324:              *     */
325:              *
326:              *     intern->u.file.delimiter = '/';
327:              *     intern->u.file.enclosure = '';
328:              *     intern->u.file.escape = '\\';
329:              *
330:              *     intern->u.file.func_getCurr = zend_hash_str_find_ptr(intern->std.ce->function_table, "getcurrenttime", sizeof("getcurrenttime") - 1);
331:              *
332:              *     return SUCCESS;
333:              * } /* }}} */
334:              *
335:              * /* {{{ spl_filesystem_object_clone */
336:              * /* Local zend_object creation (on stack)
337:              * Load the 'other' object
338:              * Create a new empty object (See spl_filesystem_object_new_ex)
339:              * Open the directory
340:              * Clone other members (properties)
341:              */
342:              * static zend_object *spl_filesystem_object_clone(zval *obj)
343:              * {
344:              *     zend_object *old_obj;
345:              *     zend_object *new_obj;
346:              *     spl_filesystem_object *intern;
347:              *     spl_filesystem_object *source;
348:              *     int index, skip_dots;
349:              *
350:              *     old_obj = Z_OBJ_P(obj);
351:              *     source = spl_filesystem_from_obj(old_obj);
352:              *     new_obj = spl_filesystem_object_new_ex(old_obj->ce);
353:              *     intern = spl_filesystem_from_obj(new_obj);
354:              *
355:              *     intern->flags = source->flags;
356:              *
357:              *     switch (source->type) {
358:              *     case SPL_FS_DIR:
359:              *         intern->u.path_len = source->u.path_len;
360:              *         intern->u.path = estrndup(source->u.path, source->u.path_len);
361:              *         intern->u.file_name_len = source->u.file_name_len;
362:              *         intern->u.file_name = estrndup(source->u.file_name, intern->u.file_name_len);
363:              *         break;
364:              *     case SPL_FS_FILE:
365:              *         spl_filesystem_dir_open(intern, source->u.path);
366:              *         /* read until we hit the position in which we were before */
367:              *         skip_dots = SPL_HAS_FLAG(source->flags, SPL_FILE_DIR_SKIPDOTS);
368:              *         for (index = 0; index < source->u.dir.index; ++index) {
369:              *             do {
370:              *                 spl_filesystem_dir_read(intern);
371:              *                 while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry_d_name))
372:              *                     continue;
373:              *             } while (1);
374:              *             intern->u.dir.index = index;
375:              *             break;
376:              *         case SPL_FS_FILE:
377:

```

```

376:     zend_throw_error(NULL, "An object of class is cannot be cloned", ZSTR_VAL(oid_object->ce->name));
377:     return new_object;
378: }
379:
380: intern->file_class = source->file_class;
381: intern->info_class = source->info_class;
382: intern->path = source->path;
383: intern->path_handler = source->path_handler;
384:
385: zend_object_clone_members(new_object, oid_object);
386:
387: IF (intern->path_handler && intern->path_handler->clone) {
388:     intern->path_handler->clone(source, intern);
389: }
390:
391: return new_object;
392: }
393: /* }}} */
394:
395: void spl_filesystem_info_set_filename(spl_filesystem_object *intern, char *path, size_t len, size_t use_copy) /* {{{ */
396: {
397:     char *p1, *p2;
398:
399:     IF (intern->file_name) {
400:         efree(intern->file_name);
401:     }
402:
403:     intern->file_name = use_copy ? estrndup(path, len) : path;
404:     intern->file_name_len = len;
405:
406:     while (intern->file_name_len > 1 && IS_SLASH_AT(intern->file_name, intern->file_name_len-1)) {
407:         intern->file_name[intern->file_name_len-1] = 0;
408:         intern->file_name_len--;
409:     }
410:
411:     p1 = strchr(intern->file_name, '/');
412:     IF (defined(PHP_WIN32))
413:         p2 = strchr(intern->file_name, '\\');
414:     else
415:         p2 = 0;
416:     IF (p1 || p2) {
417:         intern->path_len = ((p1 > p2 ? p1 : p2) - intern->file_name);
418:     } else {
419:         intern->path_len = 0;
420:     }
421:
422:     IF (intern->path) {
423:         efree(intern->path);
424:     }
425:
426:     intern->path = estrndup(path, intern->path_len);
427: } /* }}} */
428:
429: static spl_filesystem_object *spl_filesystem_object_create_info(spl_filesystem_object *source, char *file_path, size_t file_path_len, int use_copy, zend_class_entry *ce, zval *return_value) /* {{{ */
430: {
431:     spl_filesystem_object *intern;
432:     zval arg1;
433:     zend_error_handling error_handling;
434:
435:     IF (file_path || file_path_len) {
436:         IF (defined(PHP_WIN32))
437:             zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Cannot create SplFileInfo for empty path");
438:         IF (file_path && !use_copy) {
439:             efree(file_path);
440:         }
441:         else
442:             IF (file_path && !use_copy) {
443:                 efree(file_path);
444:             }
445:             file_path_len = 1;
446:             file_path = "/";
447:         IF (defined(PHP_WIN32))
448:             return NULL;
449:     }
450:
451:     zend_replace_error_handling(EH_THROW, spl_ce_RuntimeException, error_handling);
452:
453:     ce = ce ? ce : source->info_class;
454:
455:     zend_update_class_constants(ce);
456:
457:     intern = spl_filesystem_from_obj(spl_filesystem_object_new_ex(ce));
458:     ZVAL_OBJ(return_value, &intern->std);
459:
460:     IF (ce->constructor->common.scope != spl_ce_SplFileInfo) {
461:         ZVAL_STRING(&arg1, file_path, file_path_len);
462:         zend_call_method_with_1_param(return_value, ce, ce->constructor, "__construct", NULL, &arg1);
463:         zval_ptr_dtor(&arg1);
464:     } else {
465:         spl_filesystem_info_set_filename(intern, file_path, file_path_len, use_copy);
466:     }
467:
468:     zend_restore_error_handling(error_handling);
469:     return intern;
470: } /* }}} */
471:
472: static spl_filesystem_object *spl_filesystem_object_create_type(int ht, spl_filesystem_object *source, int type, zend_class_entry *ce, zval *return_value) /* {{{ */
473: {
474:     spl_filesystem_object *intern;
475:     zend_bool use_include_path = 0;
476:     zval arg1, arg2;
477:     zend_error_handling error_handling;
478:
479:     zend_replace_error_handling(EH_THROW, spl_ce_RuntimeException, error_handling);
480:
481:     switch (source->type) {
482:         case SPL_FS_THROW:
483:             case SPL_FS_FILE:
484:                 break;
485:             case SPL_FS_DIR:
486:                 IF (!source->u.dir.entry.d_name[0]) {
487:                     zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Could not open file");
488:                     zend_restore_error_handling(error_handling);
489:                     return NULL;
490:                 }
491:             }
492:
493:     switch (type) {
494:         case SPL_FS_THROW:
495:             ce = ce ? ce : source->info_class;
496:
497:             IF (UNEXPECTED(zend_update_class_constants(ce) != SUCCESS)) {
498:                 break;
499:             }
500:
501:             intern = spl_filesystem_from_obj(spl_filesystem_object_new_ex(ce));
502:             ZVAL_OBJ(return_value, &intern->std);
503:
504:             spl_filesystem_object_get_file_name(source);
505:             IF (ce->constructor->common.scope != spl_ce_SplFileInfo) {
506:                 ZVAL_STRING(&arg1, source->file_name, source->file_name_len);
507:                 zend_call_method_with_1_param(return_value, ce, ce->constructor, "__construct", NULL, &arg1);
508:                 zval_ptr_dtor(&arg1);
509:             } else {
510:                 intern->file_name = estrndup(source->file_name, source->file_name_len);
511:                 intern->file_name_len = source->file_name_len;
512:                 intern->path = spl_filesystem_object_get_path(source, &intern->path_len);
513:                 intern->path = estrndup(intern->path, intern->path_len);
514:             }
515:             break;
516:             case SPL_FS_FILE:
517:                 ce = ce ? ce : source->file_class;
518:
519:                 IF (UNEXPECTED(zend_update_class_constants(ce) != SUCCESS)) {
520:                     break;
521:                 }
522:
523:                 intern = spl_filesystem_from_obj(spl_filesystem_object_new_ex(ce));
524:                 ZVAL_OBJ(return_value, &intern->std);
525:
526:                 spl_filesystem_object_get_file_name(source);
527:
528:                 IF (ce->constructor->common.scope != spl_ce_SplFileInfo) {
529:                     ZVAL_STRING(&arg1, source->file_name, source->file_name_len);
530:                     ZVAL_STRING(&arg2, "r", 1);
531:                     zend_call_method_with_2_param(return_value, ce, ce->constructor, "__construct", NULL, &arg1, &arg2);
532:                     zval_ptr_dtor(&arg1);
533:                     zval_ptr_dtor(&arg2);
534:                 } else {
535:                     intern->file_name = source->file_name;
536:                     intern->file_name_len = source->file_name_len;
537:                     intern->path = spl_filesystem_object_get_path(source, &intern->path_len);
538:                     intern->path = estrndup(intern->path, intern->path_len);
539:
540:                     intern->u.file.open_mode = "r";
541:                     intern->u.file.open_mode_len = 1;
542:
543:                     IF (ht && zend_parse_parameters(ht, "dbr",
544:                         &intern->u.file.open_mode, &intern->u.file.open_mode_len,
545:                         &use_include_path, &intern->u.file.construct == FAILURE)) {
546:                         zend_restore_error_handling(error_handling);
547:                         intern->u.file.open_mode = NULL;
548:                         intern->file_name = NULL;
549:                         zval_ptr_dtor(return_value);
550:                         ZVAL_NULL(return_value);
551:                         return NULL;
552:                     }
553:                 }
554:
555:                 IF (spl_filesystem_file_open(intern, use_include_path, 0) == FAILURE) {
556:                     zend_restore_error_handling(error_handling);
557:                     zval_ptr_dtor(return_value);
558:                     ZVAL_NULL(return_value);
559:                     return NULL;
560:                 }
561:             }
562:         }
563:     }
564: }
565:
566:
567:
568:
569:
570:
571:
572:
573:
574:
575:
576:
577:
578:
579:
580:
581:
582:
583:
584:
585:
586:
587:
588:
589:
590:
591:
592:
593:
594:
595:
596:
597:
598:
599:
600:
601:
602:
603:
604:
605:
606:
607:
608:
609:
610:
611:
612:
613:
614:
615:
616:
617:
618:
619:
620:
621:
622:
623:
624:
625:
626:
627:
628:
629:
630:
631:
632:
633:
634:
635:
636:
637:
638:
639:
640:
641:
642:
643:
644:
645:
646:
647:
648:
649:
650:
651:
652:
653:
654:
655:
656:
657:
658:
659:
660:
661:
662:
663:
664:
665:
666:
667:
668:
669:
670:
671:
672:
673:
674:
675:
676:
677:
678:
679:
680:
681:
682:
683:
684:
685:
686:
687:
688:
689:
690:
691:
692:
693:
694:
695:
696:
697:
698:
699:
700:
701:
702:
703:
704:
705:
706:
707:
708:
709:
710:
711:
712:
713:
714:
715:
716:
717:
718:
719:
720:
721:
722:
723:
724:
725:
726:
727:
728:
729:
730:
731:
732:
733:
734:
735:
736:
737:
738:
739:
740:
741:
742:
743:
744:
745:
746:
747:
748:
749:
750:
751:
752:
753:
754:
755:
756:
757:
758:
759:
760:
761:
762:
763:
764:
765:
766:
767:
768:
769:
770:
771:
772:
773:
774:
775:
776:
777:
778:
779:
780:
781:
782:
783:
784:
785:
786:
787:
788:
789:
790:
791:
792:
793:
794:
795:
796:
797:
798:
799:
800:
801:
802:
803:
804:
805:
806:
807:
808:
809:
810:
811:
812:
813:
814:
815:
816:
817:
818:
819:
820:
821:
822:
823:
824:
825:
826:
827:
828:
829:
830:
831:
832:
833:
834:
835:
836:
837:
838:
839:
840:
841:
842:
843:
844:
845:
846:
847:
848:
849:
850:
851:
852:
853:
854:
855:
856:
857:
858:
859:
860:
861:
862:
863:
864:
865:
866:
867:
868:
869:
870:
871:
872:
873:
874:
875:
876:
877:
878:
879:
880:
881:
882:
883:
884:
885:
886:
887:
888:
889:
890:
891:
892:
893:
894:
895:
896:
897:
898:
899:
900:
901:
902:
903:
904:
905:
906:
907:
908:
909:
910:
911:
912:
913:
914:
915:
916:
917:
918:
919:
920:
921:
922:
923:
924:
925:
926:
927:
928:
929:
930:
931:
932:
933:
934:
935:
936:
937:
938:
939:
940:
941:
942:
943:
944:
945:
946:
947:
948:
949:
950:
951:
952:
953:
954:
955:
956:
957:
958:
959:
960:
961:
962:
963:
964:
965:
966:
967:
968:
969:
970:
971:
972:
973:
974:
975:
976:
977:
978:
979:
980:
981:
982:
983:
984:
985:
986:
987:
988:
989:
990:
991:
992:
993:
994:
995:
996:
997:
998:
999:

```



```

750: }
751: /* }}} */
752:
753: /* {{{ proto void DirectoryIterator::__construct (string path)
754:  * Constructs a new dir iterator from a path. */
755: SPL_METHOD(DirectoryIterator, __construct)
756: {
757:     spl_filesystem_object_construct (INTERNAL_FUNCTION_PARAM_PASSTHRU, 0);
758: }
759: /* }}} */
760:
761: /* {{{ proto void DirectoryIterator::rewind()
762:  * Rewind dir back to the start */
763: SPL_METHOD(DirectoryIterator, rewind)
764: {
765:     spl_filesystem_object_construct ('intern = Z_SPL_FILESYSTEM_P (getThis());
766: }
767: IF (zend_parse_parameters_none() == FAILURE) {
768:     return;
769: }
770:
771: intern->u.dir.index = 0;
772: IF (intern->u.dir.dirp) {
773:     php_stream_rewinddir (intern->u.dir.dirp);
774: }
775: spl_filesystem_dir_read (intern);
776: }
777: /* }}} */
778:
779: /* {{{ proto string DirectoryIterator::key()
780:  * Return current dir entry */
781: SPL_METHOD(DirectoryIterator, key)
782: {
783:     spl_filesystem_object_construct ('intern = Z_SPL_FILESYSTEM_P (getThis());
784: }
785: IF (zend_parse_parameters_none() == FAILURE) {
786:     return;
787: }
788:
789: IF (intern->u.dir.dirp) {
790:     RETURN_LONG (intern->u.dir.index);
791: } ELSE {
792:     RETURN_FALSE;
793: }
794: }
795: /* }}} */
796:
797: /* {{{ proto DirectoryIterator DirectoryIterator::current()
798:  * Return this (needed for Iterator interface) */
799: SPL_METHOD(DirectoryIterator, current)
800: {
801:     IF (zend_parse_parameters_none() == FAILURE) {
802:         return;
803:     }
804:     ZVAL_OBJ (&return_value, Z_OBJ_P (getThis()));
805:     Z_ADDREF_P (&return_value);
806: }
807: /* }}} */
808:
809: /* {{{ proto void DirectoryIterator::next()
810:  * Move to next entry */
811: SPL_METHOD(DirectoryIterator, next)
812: {
813:     spl_filesystem_object_construct ('intern = Z_SPL_FILESYSTEM_P (getThis());
814:     int skip_dots = SPL_BAS_FLAG (intern->flags, SPL_FILE_DIR_SKIPDOTS);
815:     IF (zend_parse_parameters_none() == FAILURE) {
816:         return;
817:     }
818: }
819:
820: intern->u.dir.index++;
821: do {
822:     spl_filesystem_dir_read (intern);
823: } while (skip_dots && spl_filesystem_is_dot (intern->u.dir.entry_d_name));
824: IF (intern->u.dir.name) {
825:     *name (intern->u.dir.name);
826:     intern->u.dir.name = NULL;
827: }
828: }
829: /* }}} */
830:
831: /* {{{ proto void DirectoryIterator::seek (int position)
832:  * Seek to the given position */
833: SPL_METHOD(DirectoryIterator, seek)
834: {
835:     spl_filesystem_object_construct ('intern = Z_SPL_FILESYSTEM_P (getThis());
836:     zval retval;
837:     zend_long pos;
838:     IF (zend_parse_parameters (ZEND_NUM_ARGS(), "l", &pos) == FAILURE) {
839:         return;
840:     }
841: }
842:
843: IF (intern->u.dir.index > pos) {
844:     /* we first rewind */
845:     zend_call_method_with_0_params (&EX (This), Z_OBJCE (EX (This)), &intern->u.dir.func_rewind, "rewind", NULL);
846: }
847:
848: while (intern->u.dir.index < pos) {
849:     int valid = 0;
850:     zend_call_method_with_0_params (&EX (This), Z_OBJCE (EX (This)), &intern->u.dir.func_valid, "valid", &retval);
851:     IF (!Z_ISSETREF (&retval)) {
852:         valid = zend_is_true (&retval);
853:         zval_get_ztor (&retval);
854:     }
855:     IF (!valid) {
856:         zend_throw_exception_ex (spl_ce_OutOfBoundsException, 0, "Seek position '%ZEND_LONG_FMT%' is out of range", pos);
857:         return;
858:     }
859:     zend_call_method_with_0_params (&EX (This), Z_OBJCE (EX (This)), &intern->u.dir.func_next, "next", NULL);
860: }
861: /* }}} */
862:
863: /* {{{ proto string DirectoryIterator::valid()
864:  * Check whether dir contains more entries */
865: SPL_METHOD(DirectoryIterator, valid)
866: {
867:     spl_filesystem_object_construct ('intern = Z_SPL_FILESYSTEM_P (getThis());
868: }
869: IF (zend_parse_parameters_none() == FAILURE) {
870:     return;
871: }
872:
873: RETURN_BOOL (intern->u.dir.entry_d_name[0] != '\0');
874: }
875: /* }}} */
876:
877: /* {{{ proto string SplFileInfo::getPath()
878:  * Return the path */
879: SPL_METHOD(SplFileInfo, getPath)
880: {
881:     spl_filesystem_object_construct ('intern = Z_SPL_FILESYSTEM_P (getThis());
882:     char *path;
883:     size_t path_len;
884:     IF (zend_parse_parameters_none() == FAILURE) {
885:         return;
886:     }
887: }
888:
889: path = spl_filesystem_object_get_path (intern, &path_len);
890: RETURN_STRING (path, path_len);
891: }
892: /* }}} */
893:
894: /* {{{ proto string SplFileInfo::getFilename()
895:  * Return filename only */
896: SPL_METHOD(SplFileInfo, getFilename)
897: {
898:     spl_filesystem_object_construct ('intern = Z_SPL_FILESYSTEM_P (getThis());
899:     size_t path_len;
900:     IF (zend_parse_parameters_none() == FAILURE) {
901:         return;
902:     }
903: }
904:
905: spl_filesystem_object_get_path (intern, &path_len);
906:
907: IF (path_len < path_len < intern->u.dir.name_len) {
908:     RETURN_STRING (intern->u.dir.name + path_len + 1, intern->u.dir.name_len - (path_len + 1));
909: } ELSE {
910:     RETURN_STRING (intern->u.dir.name, intern->u.dir.name_len);
911: }
912: }
913: /* }}} */
914:
915: /* {{{ proto string DirectoryIterator::getFilename()
916:  * Return filename of current dir entry */
917: SPL_METHOD(DirectoryIterator, getFilename)
918: {
919:     spl_filesystem_object_construct ('intern = Z_SPL_FILESYSTEM_P (getThis());
920: }
921: IF (zend_parse_parameters_none() == FAILURE) {
922:     return;
923: }
924:
925: RETURN_STRING (intern->u.dir.entry_d_name);
926: }
927: /* }}} */
928:
929: /* {{{ proto string SplFileInfo::getExtension()
930:  * Return file extension component of path */
931: SPL_METHOD(SplFileInfo, getExtension)
932: {
933:     spl_filesystem_object_construct ('intern = Z_SPL_FILESYSTEM_P (getThis());
934:     char *fname = NULL;
935:     const char *p;
936:     size_t flen;
937:     size_t path_len;

```

```

938:     size_t idx;
939:     zend_string *ret;
940:     IF (zend_parse_parameters_none() == FAILURE) {
941:         return;
942:     }
943: }
944:
945: spl_filesystem_object_get_path (intern, &path_len);
946:
947: IF (path_len < path_len < intern->u.dir.name_len) {
948:     fname = intern->u.dir.name + path_len + 1;
949:     flen = intern->u.dir.name_len - (path_len + 1);
950: } ELSE {
951:     fname = intern->u.dir.name;
952:     flen = intern->u.dir.name_len;
953: }
954:
955: ret = php_basename (fname, flen, NULL, 0);
956:
957: p = zend_search (ZSTR_VAL (ret), '.', ZSTR_LEN (ret));
958: IF (p) {
959:     idx = p - ZSTR_VAL (ret);
960:     RETVAL_STRING (ZSTR_VAL (ret) + idx + 1, ZSTR_LEN (ret) - idx - 1);
961:     zend_string_release (ret);
962:     return;
963: } ELSE {
964:     zend_string_release (ret);
965:     RETURN_EMPTY_STRING ();
966: }
967: }
968: /* }}} */
969:
970: /* {{{ proto string DirectoryIterator::getExtension()
971:  * Return the file extension component of path */
972: SPL_METHOD(DirectoryIterator, getExtension)
973: {
974:     spl_filesystem_object_construct ('intern = Z_SPL_FILESYSTEM_P (getThis());
975:     const char *p;
976:     size_t idx;
977:     zend_string *fname;
978:     IF (zend_parse_parameters_none() == FAILURE) {
979:         return;
980:     }
981: }
982:
983: fname = php_basename (intern->u.dir.entry_d_name, strlen (intern->u.dir.entry_d_name), NULL, 0);
984:
985: p = zend_search (ZSTR_VAL (fname), '.', ZSTR_LEN (fname));
986: IF (p) {
987:     idx = p - ZSTR_VAL (fname);
988:     RETVAL_STRING (ZSTR_VAL (fname) + idx + 1, ZSTR_LEN (fname) - idx - 1);
989:     zend_string_release (fname);
990:     return;
991: } ELSE {
992:     zend_string_release (fname);
993:     RETURN_EMPTY_STRING ();
994: }
995: /* }}} */
996:
997: /* {{{ proto string SplFileInfo::getBasename (string suffix)
998:  * Return filename component of path */
999: SPL_METHOD(SplFileInfo, getBasename)
1000: {
1001:     spl_filesystem_object_construct ('intern = Z_SPL_FILESYSTEM_P (getThis());
1002:     char *fname, *suffix = 0;
1003:     size_t flen;
1004:     size_t slen = 0, path_len;
1005:     IF (zend_parse_parameters (ZEND_NUM_ARGS(), "s", &suffix, &slen) == FAILURE) {
1006:         return;
1007:     }
1008: }
1009:
1010: spl_filesystem_object_get_path (intern, &path_len);
1011:
1012: IF (path_len < path_len < intern->u.dir.name_len) {
1013:     fname = intern->u.dir.name + path_len + 1;
1014:     flen = intern->u.dir.name_len - (path_len + 1);
1015: } ELSE {
1016:     fname = intern->u.dir.name;
1017:     flen = intern->u.dir.name_len;
1018: }
1019:
1020: RETURN_STR (php_basename (fname, flen, suffix, slen));
1021: }
1022: /* }}} */
1023:
1024: /* {{{ proto string DirectoryIterator::getBasename (string suffix)
1025:  * Return filename component of current dir entry */
1026: SPL_METHOD(DirectoryIterator, getBasename)
1027: {
1028:     spl_filesystem_object_construct ('intern = Z_SPL_FILESYSTEM_P (getThis());
1029:     char *suffix = 0;
1030:     size_t slen = 0;
1031:     zend_string *fname;
1032:     IF (zend_parse_parameters (ZEND_NUM_ARGS(), "s", &suffix, &slen) == FAILURE) {
1033:         return;
1034:     }
1035: }
1036:
1037: fname = php_basename (intern->u.dir.entry_d_name, strlen (intern->u.dir.entry_d_name), suffix, slen);
1038:
1039: RETVAL_STR (fname);
1040: }
1041: /* }}} */
1042:
1043: /* {{{ proto string SplFileInfo::getPathname()
1044:  * Return path and filename */
1045: SPL_METHOD(SplFileInfo, getPathname)
1046: {
1047:     spl_filesystem_object_construct ('intern = Z_SPL_FILESYSTEM_P (getThis());
1048:     char *path;
1049:     size_t path_len;
1050:     IF (zend_parse_parameters_none() == FAILURE) {
1051:         return;
1052:     }
1053: }
1054:
1055: path = spl_filesystem_object_get_pathname (intern, &path_len);
1056: IF (path != NULL) {
1057:     RETURN_STRING (path, path_len);
1058: } ELSE {
1059:     RETURN_FALSE;
1060: }
1061: /* }}} */
1062:
1063: /* {{{ proto string FilesystemIterator::key()
1064:  * Return getFilename() or getFileInfo() depending on flags */
1065: SPL_METHOD(FilesystemIterator, key)
1066: {
1067:     spl_filesystem_object_construct ('intern = Z_SPL_FILESYSTEM_P (getThis());
1068: }
1069: IF (zend_parse_parameters_none() == FAILURE) {
1070:     return;
1071: }
1072:
1073: IF (SPL_FILE_DIR_KEY (intern, SPL_FILE_DIR_KEY_AS_FILENAME)) {
1074:     RETURN_STRING (intern->u.dir.entry_d_name);
1075: } ELSE {
1076:     spl_filesystem_object_get_file_name (intern);
1077:     RETURN_STRING (intern->u.dir.name, intern->u.dir.name_len);
1078: }
1079: }
1080: /* }}} */
1081:
1082: /* {{{ proto string FilesystemIterator::current()
1083:  * Return getFilename(), getFileInfo() or this depending on flags */
1084: SPL_METHOD(FilesystemIterator, current)
1085: {
1086:     spl_filesystem_object_construct ('intern = Z_SPL_FILESYSTEM_P (getThis());
1087: }
1088: IF (zend_parse_parameters_none() == FAILURE) {
1089:     return;
1090: }
1091:
1092: IF (SPL_FILE_DIR_CURRENT (intern, SPL_FILE_DIR_CURRENT_AS_PATHNAME)) {
1093:     spl_filesystem_object_get_file_name (intern);
1094:     RETURN_STRING (intern->u.dir.name, intern->u.dir.name_len);
1095: } ELSE IF (SPL_FILE_DIR_CURRENT (intern, SPL_FILE_DIR_CURRENT_AS_FILEINFO)) {
1096:     spl_filesystem_object_get_file_name (intern);
1097:     spl_filesystem_object_get_cwata_type (0, intern, SPL_FS_INFO, NULL, &return_value);
1098:     } ELSE {
1099:     ZVAL_OBJ (&return_value, Z_OBJ_P (getThis()));
1100:     Z_ADDREF_P (&return_value);
1101:     /* RETURN_STRING (intern->u.dir.entry_d_name, 1); */
1102: }
1103: }
1104: /* }}} */
1105:
1106: /* {{{ proto bool DirectoryIterator::isDot()
1107:  * Return true if current entry is '.' or '..' */
1108: SPL_METHOD(DirectoryIterator, isDot)
1109: {
1110:     spl_filesystem_object_construct ('intern = Z_SPL_FILESYSTEM_P (getThis());
1111: }
1112: IF (zend_parse_parameters_none() == FAILURE) {
1113:     return;
1114: }
1115:
1116: RETURN_BOOL (spl_filesystem_is_dot (intern->u.dir.entry_d_name));
1117: }
1118: /* }}} */
1119:
1120: /* {{{ proto void SplFileInfo::__construct (string filename)
1121:  * Constructs a new SplFileInfo from a path */
1122: /* When the constructor gets called the object is already created
1123:  * by the engine, so we must only call 'additional' initializations.
1124:  */
1125: SPL_METHOD(SplFileInfo, __construct)

```

```

1126: {
1127:     spl_filesystem_object *intern;
1128:     char *path;
1129:     size_t len;
1130:
1131:     if (zend_parse_parameters_throw(ZEND_NUM_ARGS(), "s", &path, &len) == FAILURE) {
1132:         return;
1133:     }
1134:
1135:     intern = _spl_filesystem_P(getThis());
1136:
1137:     spl_filesystem_info_set_filename(intern, path, len, 1);
1138:
1139:     /* intern->type = SPL_FS_INFO already set */
1140:
1141:     /* {{{ */
1142:
1143:     /* {{{ FileinfoFunction */
1144:     FileinfoFunction(func_name, func_num) {
1145:         SPL_METHOD(splFileInfo, func_name) \
1146:         {
1147:             spl_filesystem_object *intern = _spl_filesystem_P(getThis());
1148:             zend_error_handling error_handling;
1149:             if (zend_parse_parameters_none() == FAILURE) {
1150:                 return;
1151:             }
1152:
1153:             zend_replace_error_handling(EH_THROW, spl_ce_RuntimeException, &error_handling);
1154:             spl_filesystem_object_get_file_name(intern);
1155:             php_stat(intern->file_name, intern->file_name_len, func_num, return_value);
1156:             zend_restore_error_handling(&error_handling);
1157:         }
1158:     } /* }}} */
1159:
1160:     /* {{{ proto int splFileInfo:getPerms()
1161:      * Get file permissions */
1162:     FileinfoFunction(getPerms, FS_PERMS)
1163:     /* }}} */
1164:
1165:     /* {{{ proto int splFileInfo:getInode()
1166:      * Get file inode */
1167:     FileinfoFunction(getInode, FS_INODE)
1168:     /* }}} */
1169:
1170:     /* {{{ proto int splFileInfo:getSize()
1171:      * Get file size */
1172:     FileinfoFunction(getSize, FS_SIZE)
1173:     /* }}} */
1174:
1175:     /* {{{ proto int splFileInfo:getOwner()
1176:      * Get file owner */
1177:     FileinfoFunction(getOwner, FS_OWNER)
1178:     /* }}} */
1179:
1180:     /* {{{ proto int splFileInfo:getGroup()
1181:      * Get file group */
1182:     FileinfoFunction(getGroup, FS_GROUP)
1183:     /* }}} */
1184:
1185:     /* {{{ proto int splFileInfo:getAtime()
1186:      * Get last access time of file */
1187:     FileinfoFunction(getAtime, FS_ATIME)
1188:     /* }}} */
1189:
1190:     /* {{{ proto int splFileInfo:getMTime()
1191:      * Get last modification time of file */
1192:     FileinfoFunction(getMTime, FS_MTIME)
1193:     /* }}} */
1194:
1195:     /* {{{ proto int splFileInfo:getCtime()
1196:      * Get inode modification time of file */
1197:     FileinfoFunction(getCtime, FS_CTIME)
1198:     /* }}} */
1199:
1200:     /* {{{ proto string splFileInfo:getType()
1201:      * Get file type */
1202:     FileinfoFunction(getType, FS_TYPE)
1203:     /* }}} */
1204:
1205:     /* {{{ proto bool splFileInfo:isWritable()
1206:      * Returns true if file can be written */
1207:     FileinfoFunction(isWritable, FS_IS_W)
1208:     /* }}} */
1209:
1210:     /* {{{ proto bool splFileInfo:isReadable()
1211:      * Returns true if file can be read */
1212:     FileinfoFunction(isReadable, FS_IS_R)
1213:     /* }}} */
1214:
1215:     /* {{{ proto bool splFileInfo:isExecutable()
1216:      * Returns true if file is executable */
1217:     FileinfoFunction(isExecutable, FS_IS_X)
1218:     /* }}} */
1219:
1220:     /* {{{ proto bool splFileInfo:isFile()
1221:      * Returns true if file is a regular file */
1222:     FileinfoFunction(isFile, FS_IS_FILE)
1223:     /* }}} */
1224:
1225:     /* {{{ proto bool splFileInfo:isDir()
1226:      * Returns true if file is directory */
1227:     FileinfoFunction(isDir, FS_IS_DIR)
1228:     /* }}} */
1229:
1230:     /* {{{ proto bool splFileInfo:isLink()
1231:      * Returns true if file is symbolic link */
1232:     FileinfoFunction(isLink, FS_IS_LINK)
1233:     /* }}} */
1234:
1235:     /* {{{ proto string splFileInfo:getLinkTarget()
1236:      * Return the target of a symbolic link */
1237:     SPL_METHOD(splFileInfo, getLinkTarget)
1238:     {
1239:         spl_filesystem_object *intern = _spl_filesystem_P(getThis());
1240:         symlink &ret;
1241:         char buff[MAXPATHLEN];
1242:         zend_error_handling error_handling;
1243:
1244:         if (zend_parse_parameters_none() == FAILURE) {
1245:             return;
1246:         }
1247:
1248:         zend_replace_error_handling(EH_THROW, spl_ce_RuntimeException, &error_handling);
1249:
1250:         #if defined(PHP_WIN32) || HAVE_SYMLINK
1251:         if (intern->file_name == NULL) {
1252:             php_error_docref(NULL, E_WARNING, "Empty filename");
1253:             RETURN_FALSE;
1254:         } else if (!IS_ABSOLUTE_PATH(intern->file_name, intern->file_name_len)) {
1255:             char expanded_path[MAXPATHLEN];
1256:             if (expand_filepath_with_mode(intern->file_name, expanded_path, NULL, 0, CHD_EXPAND)) {
1257:                 php_error_docref(NULL, E_WARNING, "No such file or directory");
1258:                 RETURN_FALSE;
1259:             }
1260:             ret = php_sya_readlink(expanded_path, buff, MAXPATHLEN - 1);
1261:         } else {
1262:             ret = php_sya_readlink(intern->file_name, buff, MAXPATHLEN - 1);
1263:         }
1264:         return ret;
1265:     } /* always fail if not implemented */
1266: #endif
1267:
1268:     if (ret == -1) {
1269:         zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Unable to read link %s, error: %s", intern->file_name, strerror(errno));
1270:         RETURN_FALSE;
1271:     } else {
1272:         /* Append NULL to the end of the string */
1273:         buff[ret] = '\0';
1274:
1275:         RETVAL_STRING(buff, ret);
1276:     }
1277:
1278:     zend_restore_error_handling(&error_handling);
1279: } /* }}} */
1280:
1281: #if HAVE_REALPATH || defined(UTS)
1282: /* {{{ proto string splFileInfo:getRealPath()
1283:  * Return the resolved path */
1284: SPL_METHOD(splFileInfo, getRealPath)
1285: {
1286:     spl_filesystem_object *intern = _spl_filesystem_P(getThis());
1287:     char buff[MAXPATHLEN];
1288:     char *filename;
1289:     zend_error_handling error_handling;
1290:
1291:     if (zend_parse_parameters_none() == FAILURE) {
1292:         return;
1293:     }
1294:
1295:     zend_replace_error_handling(EH_THROW, spl_ce_RuntimeException, &error_handling);
1296:
1297:     if (intern->type == SPL_FS_DIR && !intern->file_name && intern->u.dir.entry_dname[0]) {
1298:         spl_filesystem_object_get_file_name(intern);
1299:     }
1300:
1301:     if (intern->orig_path) {
1302:         filename = intern->orig_path;
1303:     } else {
1304:         filename = intern->file_name;
1305:     }
1306:
1307:     if (filename && !VOWD_REALPATH(filename, buff)) {
1308:         #if defined(UTS)
1309:         if (filename && VOWD_REALPATH(filename, buff)) {
1310:             #endif
1311:             if (VOWD_ACCESS(buff, F_OK)) {
1312:                 RETURN_FALSE;
1313:             } else

```

```

1314: #endif
1315:         RETVAL_STRING(buff);
1316:     } else {
1317:         RETURN_FALSE;
1318:     }
1319:
1320:     zend_restore_error_handling(&error_handling);
1321: }
1322: /* }}} */
1323: #endif
1324:
1325: /* {{{ proto SplFileInfo:SplFileInfo(openFile(string mode = "r" [, bool use_include_path [, resource context]])
1326:  * Open the current file */
1327: SPL_METHOD(splFileInfo, openFile)
1328: {
1329:     spl_filesystem_object *intern = _spl_filesystem_P(getThis());
1330:
1331:     spl_filesystem_object_create_type(ZEND_NUM_ARGS(), intern, SPL_FS_FILE, NULL, return_value);
1332: }
1333: /* }}} */
1334:
1335: /* {{{ proto void SplFileInfo:setFileClass(string class_name)
1336:  * Class to use in openFile() */
1337: SPL_METHOD(splFileInfo, setFileClass)
1338: {
1339:     spl_filesystem_object *intern = _spl_filesystem_P(getThis());
1340:     zend_class_entry *ce = spl_ce_SplFileInfo;
1341:     zend_error_handling error_handling;
1342:
1343:     zend_replace_error_handling(EH_THROW, spl_ce_UnexpectedValueException, &error_handling);
1344:
1345:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "|C", &ce) == SUCCESS) {
1346:         intern->file_class = ce;
1347:     }
1348:
1349:     zend_restore_error_handling(&error_handling);
1350: }
1351: /* }}} */
1352:
1353: /* {{{ proto void SplFileInfo:setInfoClass(string class_name)
1354:  * Class to use in getFileInfo(), getPathInfo() */
1355: SPL_METHOD(splFileInfo, setInfoClass)
1356: {
1357:     spl_filesystem_object *intern = _spl_filesystem_P(getThis());
1358:     zend_class_entry *ce = spl_ce_SplFileInfo;
1359:     zend_error_handling error_handling;
1360:
1361:     zend_replace_error_handling(EH_THROW, spl_ce_UnexpectedValueException, &error_handling);
1362:
1363:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "|C", &ce) == SUCCESS) {
1364:         intern->info_class = ce;
1365:     }
1366:
1367:     zend_restore_error_handling(&error_handling);
1368: }
1369: /* }}} */
1370:
1371: /* {{{ proto SplFileInfo:SplFileInfo(getFileInfo(string $class_name)
1372:  * Get/copy file info */
1373: SPL_METHOD(splFileInfo, getFileInfo)
1374: {
1375:     spl_filesystem_object *intern = _spl_filesystem_P(getThis());
1376:     zend_class_entry *ce = intern->info_class;
1377:     zend_error_handling error_handling;
1378:
1379:     zend_replace_error_handling(EH_THROW, spl_ce_UnexpectedValueException, &error_handling);
1380:
1381:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "|C", &ce) == SUCCESS) {
1382:         spl_filesystem_object_create_type(ZEND_NUM_ARGS(), intern, SPL_FS_INFO, ce, return_value);
1383:     }
1384:
1385:     zend_restore_error_handling(&error_handling);
1386: }
1387: /* }}} */
1388:
1389: /* {{{ proto SplFileInfo:SplFileInfo(getPathInfo(string $class_name)
1390:  * Get/copy file info */
1391: SPL_METHOD(splFileInfo, getPathInfo)
1392: {
1393:     spl_filesystem_object *intern = _spl_filesystem_P(getThis());
1394:     zend_class_entry *ce = intern->info_class;
1395:     zend_error_handling error_handling;
1396:
1397:     zend_replace_error_handling(EH_THROW, spl_ce_UnexpectedValueException, &error_handling);
1398:
1399:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "|C", &ce) == SUCCESS) {
1400:         char *path = spl_filesystem_object_get_pathname(intern, &path_len);
1401:         if (path) {
1402:             char *dpath = estrdup(path, path_len);
1403:             path_len = php_dirname(dpath, path_len);
1404:             spl_filesystem_object_create_info(intern, dpath, path_len, 1, ce, return_value);
1405:             efree(dpath);
1406:         }
1407:     }
1408:
1409:     zend_restore_error_handling(&error_handling);
1410: }
1411:
1412: /* }}} */
1413:
1414: /* {{{ proto void SplFileInfo::_bad_state_ex(void) */
1415: SPL_METHOD(splFileInfo, _bad_state_ex)
1416: {
1417:     zend_throw_exception_ex(spl_ce_LogicException, 0,
1418:         "The parent constructor was not called: the object is in an "
1419:         "invalid state");
1420: }
1421: /* }}} */
1422:
1423: /* {{{ proto void FilesystemIterator::__construct(string path [, int flags])
1424:  * Constructs a new dir iterator from a path. */
1425: SPL_METHOD(FilesystemIterator, __construct)
1426: {
1427:     spl_filesystem_object_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, DIT_CTOR_FLAGS | SPL_FILE_DIR_SKIPDOTS);
1428: }
1429: /* }}} */
1430:
1431: /* {{{ proto void FilesystemIterator::rewind()
1432:  * Rewind dir back to the start */
1433: SPL_METHOD(FilesystemIterator, rewind)
1434: {
1435:     spl_filesystem_object *intern = _spl_filesystem_P(getThis());
1436:     int skip_dots = SPL_HAS_FLAG(intern->flags, SPL_FILE_DIR_SKIPDOTS);
1437:
1438:     if (zend_parse_parameters_none() == FAILURE) {
1439:         return;
1440:     }
1441:
1442:     intern->u.dir.index = 0;
1443:     if (intern->u.dir.dirp) {
1444:         php_stream_rewinddd(intern->u.dir.dirp);
1445:     }
1446:
1447:     do {
1448:         spl_filesystem_dir_read(intern);
1449:     } while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry_dname));
1450: } /* }}} */
1451:
1452: /* {{{ proto int FilesystemIterator:getFlags()
1453:  * Get handling flags */
1454: SPL_METHOD(FilesystemIterator, getFlags)
1455: {
1456:     spl_filesystem_object *intern = _spl_filesystem_P(getThis());
1457:
1458:     if (zend_parse_parameters_none() == FAILURE) {
1459:         return;
1460:     }
1461:
1462:     RETURN_LONG(intern->flags & (SPL_FILE_DIR_KEY_MODE_MASK | SPL_FILE_DIR_CURRENT_MODE_MASK | SPL_FILE_DIR_OTHERS_MASK));
1463: } /* }}} */
1464:
1465: /* {{{ proto void FilesystemIterator:setFlags(long $flags)
1466:  * Set handling flags */
1467: SPL_METHOD(FilesystemIterator, setFlags)
1468: {
1469:     spl_filesystem_object *intern = _spl_filesystem_P(getThis());
1470:     zend_long flags;
1471:
1472:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "l", &flags) == FAILURE) {
1473:         return;
1474:     }
1475:
1476:     intern->flags &= (SPL_FILE_DIR_KEY_MODE_MASK | SPL_FILE_DIR_CURRENT_MODE_MASK | SPL_FILE_DIR_OTHERS_MASK);
1477:     intern->flags |= ((SPL_FILE_DIR_KEY_MODE_MASK | SPL_FILE_DIR_CURRENT_MODE_MASK | SPL_FILE_DIR_OTHERS_MASK) & flags);
1478: } /* }}} */
1479:
1480: /* {{{ proto bool RecursiveDirectoryIterator::hasChildren(bool $allow_links = false)
1481:  * Returns whether current entry is a directory and not "." or ".." */
1482: SPL_METHOD(RecursiveDirectoryIterator, hasChildren)
1483: {
1484:     zend_bool allow_links = 0;
1485:     spl_filesystem_object *intern = _spl_filesystem_P(getThis());
1486:
1487:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "|b", &allow_links) == FAILURE) {
1488:         return;
1489:     }
1490:
1491:     if (spl_filesystem_is_invalid_or_dot(intern->u.dir.entry_dname)) {
1492:         RETURN_FALSE;
1493:     } else {
1494:         spl_filesystem_object_get_file_name(intern);
1495:         if (allow_links && (intern->flags & SPL_FILE_DIR_FOLLOW_SYMLINKS)) {
1496:             php_stat(intern->file_name, intern->file_name_len, FS_IS_LINK, return_value);
1497:             if (zend_is_true(return_value)) {
1498:                 RETURN_FALSE;
1499:             }
1500:         }
1501:         php_stat(intern->file_name, intern->file_name_len, FS_IS_DIR, return_value);
1502:     }

```

```

1502: }
1503: /* }}} */
1504:
1505: /* {{{ proto RecursiveDirectoryIterator DirectoryIterator::getChildren()
1506:  * Returns an iterator for the current entry if it is a directory */
1507: SPL_METHOD(RecursiveDirectoryIterator, getChildren)
1508: {
1509:     zval $path, $flags;
1510:     spl_filesystem_object *intern = _SPL_FILESYSTEM_P(getThis());
1511:     spl_filesystem_object *subdir;
1512:     char slash = SPL_HAS_FLAG(intern->flags, SPL_FILE_DIR_UNIXPATHS) ? '/' : DEFAULT_SLASH;
1513:
1514:     IF (zend_parse_parameters_none() == FAILURE) {
1515:         return;
1516:     }
1517:
1518:     spl_filesystem_object_get_file_name(intern);
1519:
1520:     ZVAL_LONG($flags, intern->flags);
1521:     ZVAL_STRING($path, intern->file_name, intern->file_name_len);
1522:     spl_instance_init_arg_ext(2, OBJ_CEP(getThis()), return_value, $path, $flags);
1523:     zval_ptr_dtor($path);
1524:     zval_ptr_dtor($flags);
1525:
1526:     subdir = _SPL_FILESYSTEM_P(return_value);
1527:     IF (subdir) {
1528:         IF (intern->u.dir.sub_path && intern->u.dir.sub_path[0]) {
1529:             subdir->u.dir.sub_path_len = sprintf(subdir->u.dir.sub_path, 0, "%s%s", intern->u.dir.sub_path, slash, intern->u.dir.entry_d_name);
1530:         } else {
1531:             subdir->u.dir.sub_path_len = strlen(intern->u.dir.entry_d_name);
1532:             subdir->u.dir.sub_path = estrndup(intern->u.dir.entry_d_name, subdir->u.dir.sub_path_len);
1533:         }
1534:         subdir->info_class = intern->info_class;
1535:         subdir->file_class = intern->file_class;
1536:         subdir->oth = intern->oth;
1537:     }
1538:     /* }}} */
1539:     /* }}} */
1540:
1541:     /* {{{ proto void RecursiveDirectoryIterator::getSubPath()
1542:     * Get sub path */
1543:     SPL_METHOD(RecursiveDirectoryIterator, getSubPath)
1544:     {
1545:         spl_filesystem_object *intern = _SPL_FILESYSTEM_P(getThis());
1546:
1547:         IF (zend_parse_parameters_none() == FAILURE) {
1548:             return;
1549:         }
1550:
1551:         IF (intern->u.dir.sub_path) {
1552:             RETURN_STRING(intern->u.dir.sub_path, intern->u.dir.sub_path_len);
1553:         } else {
1554:             RETURN_EMPTY_STRING();
1555:         }
1556:     }
1557:     /* }}} */
1558:
1559:     /* {{{ proto void RecursiveDirectoryIterator::getSubPathname()
1560:     * Get sub path and file name */
1561:     SPL_METHOD(RecursiveDirectoryIterator, getSubPathname)
1562:     {
1563:         spl_filesystem_object *intern = _SPL_FILESYSTEM_P(getThis());
1564:         char slash = SPL_HAS_FLAG(intern->flags, SPL_FILE_DIR_UNIXPATHS) ? '/' : DEFAULT_SLASH;
1565:
1566:         IF (zend_parse_parameters_none() == FAILURE) {
1567:             return;
1568:         }
1569:
1570:         IF (intern->u.dir.sub_path) {
1571:             RETURN_NEW_STR(1, sprintf(0, "%s%s", intern->u.dir.sub_path, slash, intern->u.dir.entry_d_name));
1572:         } else {
1573:             RETURN_STRING(intern->u.dir.entry_d_name);
1574:         }
1575:     }
1576:     /* }}} */
1577:
1578:     /* {{{ proto int RecursiveDirectoryIterator::__construct(string path [, int flags])
1579:     * Constructs a new dir iterator from a path. */
1580:     SPL_METHOD(RecursiveDirectoryIterator, __construct)
1581:     {
1582:         spl_filesystem_object_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, DIT_CTOR_FLAGS);
1583:     }
1584:     /* }}} */
1585:
1586:     #ifdef HAVE_GLOB
1587:     /* {{{ proto int GlobIterator::__construct(string path [, int flags])
1588:     * Constructs a new dir iterator from a glob expression (no globlib needed). */
1589:     SPL_METHOD(GlobIterator, __construct)
1590:     {
1591:         spl_filesystem_object_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, DIT_CTOR_FLAGS|DIT_CTOR_GLOB);
1592:     }
1593:     /* }}} */
1594:
1595:     /* {{{ proto int GlobIterator::count()
1596:     * Return the number of directories and files found by globbing */
1597:     SPL_METHOD(GlobIterator, count)
1598:     {
1599:         spl_filesystem_object *intern = _SPL_FILESYSTEM_P(getThis());
1600:
1601:         IF (zend_parse_parameters_none() == FAILURE) {
1602:             return;
1603:         }
1604:
1605:         IF (intern->u.dir.dirp && php_stream_is(intern->u.dir.dirp, &php_glob_stream_ops)) {
1606:             RETURN_LONG(php_glob_stream_get_count(intern->u.dir.dirp, NULL));
1607:         } else {
1608:             /* should not happen */
1609:             php_error_docref(NULL, E_ERROR, "GlobIterator lost glob state");
1610:         }
1611:     }
1612:     /* }}} */
1613:     #endif /* HAVE_GLOB */
1614:
1615:     /* {{{ forward declarations to the iterator handlers */
1616:     static void spl_filesystem_dir_it_dtor(spl_filesystem_object *iter);
1617:     static int spl_filesystem_dir_it_valid(spl_filesystem_object *iter);
1618:     static zval *spl_filesystem_dir_it_current_data(spl_filesystem_object *iter);
1619:     static void spl_filesystem_dir_it_current_key(spl_filesystem_object *iter, zval *key);
1620:     static void spl_filesystem_dir_it_move_forward(spl_filesystem_object *iter);
1621:     static void spl_filesystem_dir_it_rewind(spl_filesystem_object *iter);
1622:
1623:     /* iterator handler table */
1624:     static const zend_object_iterator_funcs spl_filesystem_dir_it_funcs = {
1625:         spl_filesystem_dir_it_dtor,
1626:         spl_filesystem_dir_it_valid,
1627:         spl_filesystem_dir_it_current_data,
1628:         spl_filesystem_dir_it_current_key,
1629:         spl_filesystem_dir_it_move_forward,
1630:         spl_filesystem_dir_it_rewind,
1631:         NULL
1632:     };
1633:     /* }}} */
1634:
1635:     /* {{{ spl_on_dir_get_iterator */
1636:     zend_object_iterator *spl_filesystem_dir_get_iterator(zend_class_entry *ce, zval *object, int by_ref)
1637:     {
1638:         spl_filesystem_object *iter;
1639:         spl_filesystem_object *dir_object;
1640:
1641:         IF (by_ref) {
1642:             zend_throw_exception(spl_ce_RuntimeException, "An iterator cannot be used with foreach by reference", 0);
1643:             return NULL;
1644:         }
1645:
1646:         dir_object = _SPL_FILESYSTEM_P(object);
1647:         iter = spl_filesystem_object_to_iterator(dir_object);
1648:         ZVAL_COPY(&iter->intern.data, object);
1649:         iter->intern.funcs = spl_filesystem_dir_it_funcs;
1650:         /* ~current must be initialized! rewind doesn't set it and valid
1651:          * doesn't check whether it's set */
1652:         iter->current = *object;
1653:         return iter->intern;
1654:     }
1655:     /* }}} */
1656:
1657:     /* {{{ spl_filesystem_dir_it_dtor */
1658:     static void spl_filesystem_dir_it_dtor(spl_filesystem_object *iter)
1659:     {
1660:         spl_filesystem_object *iterator = (spl_filesystem_object *)iter;
1661:
1662:         IF (!IS_ISNULL(iterator->intern.data)) {
1663:             zval *obj = iterator->intern.data;
1664:             zval_ptr_dtor(obj);
1665:         }
1666:
1667:         /* Otherwise we were called from the owning object free storage handler as
1668:          * it sets iterator->intern.data to IS_UNDEF.
1669:          * We don't even need to destroy iterator->current as we didn't add a
1670:          * reference to it in move_forward or get_iterator */
1671:     }
1672:     /* }}} */
1673:
1674:     /* {{{ spl_filesystem_dir_it_valid */
1675:     static int spl_filesystem_dir_it_valid(spl_filesystem_object *iter)
1676:     {
1677:         spl_filesystem_object *object = spl_filesystem_iterator_to_object(spl_filesystem_object *iter);
1678:
1679:         return object->u.dir.entry_d_name[0] != '\0' ? SUCCESS : FAILURE;
1680:     }
1681:     /* }}} */
1682:
1683:     /* {{{ spl_filesystem_dir_it_current_data */
1684:     static zval *spl_filesystem_dir_it_current_data(spl_filesystem_object *iter)
1685:     {
1686:         spl_filesystem_object *iterator = (spl_filesystem_object *)iter;
1687:
1688:         return iterator->current;
1689:     }
1690:     /* }}} */

```

```

1691:     /* {{{ spl_filesystem_dir_it_current_key */
1692:     static void spl_filesystem_dir_it_current_key(spl_filesystem_object *iter, zval *key)
1693:     {
1694:         spl_filesystem_object *object = spl_filesystem_iterator_to_object(spl_filesystem_object *iter);
1695:
1696:         ZVAL_LONG(key, object->u.dir.index);
1697:     }
1698:     /* }}} */
1699:
1700:     /* {{{ spl_filesystem_dir_it_move_forward */
1701:     static void spl_filesystem_dir_it_move_forward(spl_filesystem_object *iter)
1702:     {
1703:         spl_filesystem_object *object = spl_filesystem_iterator_to_object(spl_filesystem_object *iter);
1704:
1705:         object->u.dir.index++;
1706:         spl_filesystem_dir_read(object);
1707:         IF (object->file_name) {
1708:             zfree(object->file_name);
1709:             object->file_name = NULL;
1710:         }
1711:     }
1712:     /* }}} */
1713:
1714:     /* {{{ spl_filesystem_dir_it_rewind */
1715:     static void spl_filesystem_dir_it_rewind(spl_filesystem_object *iter)
1716:     {
1717:         spl_filesystem_object *object = spl_filesystem_iterator_to_object(spl_filesystem_object *iter);
1718:
1719:         object->u.dir.index = 0;
1720:         IF (object->u.dir.dirp) {
1721:             php_stream_rewinddir(object->u.dir.dirp);
1722:         }
1723:         spl_filesystem_dir_read(object);
1724:     }
1725:     /* }}} */
1726:
1727:     /* {{{ spl_filesystem_tree_it_dtor */
1728:     static void spl_filesystem_tree_it_dtor(spl_filesystem_object *iter)
1729:     {
1730:         spl_filesystem_object *iterator = (spl_filesystem_object *)iter;
1731:
1732:         IF (!IS_ISNULL(iterator->intern.data)) {
1733:             zval *object = iterator->intern.data;
1734:             zval_ptr_dtor(object);
1735:         } else {
1736:             IF (!IS_ISNULL(iterator->current)) {
1737:                 zval_ptr_dtor(iterator->current);
1738:                 ZVAL_UNDEF(iterator->current);
1739:             }
1740:         }
1741:     }
1742:     /* }}} */
1743:
1744:     /* {{{ spl_filesystem_tree_it_current_data */
1745:     static zval *spl_filesystem_tree_it_current_data(spl_filesystem_object *iter)
1746:     {
1747:         spl_filesystem_object *iterator = (spl_filesystem_object *)iter;
1748:         spl_filesystem_object *object = spl_filesystem_iterator_to_object(iterator);
1749:
1750:         IF (SPL_FILE_DIR_CURRENT(object, SPL_FILE_DIR_CURRENT_AS_PATHNAME)) {
1751:             IF (!IS_ISNULL(iterator->current)) {
1752:                 spl_filesystem_object_get_file_name(object);
1753:                 ZVAL_STRING(iterator->current, object->file_name, object->file_name_len);
1754:             }
1755:             return iterator->current;
1756:         } else IF (SPL_FILE_DIR_CURRENT(object, SPL_FILE_DIR_CURRENT_AS_FILEINFO)) {
1757:             IF (!IS_ISNULL(iterator->current)) {
1758:                 spl_filesystem_object_get_file_name(object);
1759:                 spl_filesystem_object_create_type(0, object, SPL_FILE_INFO, NULL, iterator->current);
1760:             }
1761:             return iterator->current;
1762:         } else {
1763:             return iterator->intern.data;
1764:         }
1765:     }
1766:     /* }}} */
1767:
1768:     /* {{{ spl_filesystem_tree_it_current_key */
1769:     static void spl_filesystem_tree_it_current_key(spl_filesystem_object *iter, zval *key)
1770:     {
1771:         spl_filesystem_object *object = spl_filesystem_iterator_to_object(spl_filesystem_object *iter);
1772:
1773:         IF (SPL_FILE_DIR_KEY(object, SPL_FILE_DIR_KEY_AS_FILENAME)) {
1774:             ZVAL_STRING(key, object->u.dir.entry_d_name);
1775:         } else {
1776:             spl_filesystem_object_get_file_name(object);
1777:             ZVAL_STRING(key, object->file_name, object->file_name_len);
1778:         }
1779:     }
1780:     /* }}} */
1781:
1782:     /* {{{ spl_filesystem_tree_it_move_forward */
1783:     static void spl_filesystem_tree_it_move_forward(spl_filesystem_object *iter)
1784:     {
1785:         spl_filesystem_object *iterator = (spl_filesystem_object *)iter;
1786:         spl_filesystem_object *object = spl_filesystem_iterator_to_object(iterator);
1787:
1788:         object->u.dir.index++;
1789:         IF (
1790:             spl_filesystem_dir_read(object);
1791:             while (spl_filesystem_is_dot(object->u.dir.entry_d_name));
1792:             zfree(object->file_name);
1793:             object->file_name = NULL;
1794:         )
1795:         {
1796:             IF (!IS_ISNULL(iterator->current)) {
1797:                 zval_ptr_dtor(iterator->current);
1798:                 ZVAL_UNDEF(iterator->current);
1799:             }
1800:         }
1801:     }
1802:     /* }}} */
1803:
1804:     /* {{{ spl_filesystem_tree_it_rewind */
1805:     static void spl_filesystem_tree_it_rewind(spl_filesystem_object *iter)
1806:     {
1807:         spl_filesystem_object *iterator = (spl_filesystem_object *)iter;
1808:         spl_filesystem_object *object = spl_filesystem_iterator_to_object(iterator);
1809:
1810:         object->u.dir.index = 0;
1811:         IF (object->u.dir.dirp) {
1812:             php_stream_rewinddir(object->u.dir.dirp);
1813:         }
1814:         IF (
1815:             spl_filesystem_dir_read(object);
1816:             while (spl_filesystem_is_dot(object->u.dir.entry_d_name));
1817:             zval_ptr_dtor(iterator->current);
1818:             ZVAL_UNDEF(iterator->current);
1819:         )
1820:         {
1821:             /* }}} */
1822:
1823:     /* {{{ iterator handler table */
1824:     static const zend_object_iterator_funcs spl_filesystem_tree_it_funcs = {
1825:         spl_filesystem_tree_it_dtor,
1826:         spl_filesystem_dir_it_valid,
1827:         spl_filesystem_tree_it_current_data,
1828:         spl_filesystem_tree_it_current_key,
1829:         spl_filesystem_tree_it_move_forward,
1830:         spl_filesystem_tree_it_rewind,
1831:         NULL
1832:     };
1833:     /* }}} */
1834:
1835:     /* {{{ spl_on_dir_get_iterator */
1836:     zend_object_iterator *spl_filesystem_tree_get_iterator(zend_class_entry *ce, zval *object, int by_ref)
1837:     {
1838:         spl_filesystem_object *iter;
1839:         spl_filesystem_object *dir_object;
1840:
1841:         IF (by_ref) {
1842:             zend_throw_exception(spl_ce_RuntimeException, "An iterator cannot be used with foreach by reference", 0);
1843:             return NULL;
1844:         }
1845:
1846:         dir_object = _SPL_FILESYSTEM_P(object);
1847:         iter = spl_filesystem_object_to_iterator(dir_object);
1848:         ZVAL_COPY(&iter->intern.data, object);
1849:         iter->intern.funcs = spl_filesystem_tree_it_funcs;
1850:         return iter->intern;
1851:     }
1852:     /* }}} */
1853:
1854:     /* {{{ spl_filesystem_object_dtor */
1855:     static int spl_filesystem_object_dtor(zval *readobj, zval *writeobj, int type)
1856:     {
1857:         spl_filesystem_object *intern = _SPL_FILESYSTEM_P(readobj);
1858:
1859:         IF (type == IS_STRING) {
1860:             IF (IS_OBJ_CEP(readobj)->is_string) {
1861:                 return std_object_handlers.dtor_obj(readobj, writeobj, type);
1862:             }
1863:         }
1864:
1865:         switch (intern->type) {
1866:             case SPL_FS_INFO:
1867:             case SPL_FS_FILE:
1868:                 ZVAL_STRING(writeobj, intern->file_name, intern->file_name_len);
1869:                 return SUCCESS;
1870:             case SPL_FS_DIR:
1871:                 ZVAL_STRING(writeobj, intern->u.dir.entry_d_name);
1872:                 return SUCCESS;
1873:         }
1874:         /* else if (type == IS_BOOL) {
1875:             ZVAL_TRUE(writeobj);
1876:             return SUCCESS;
1877:         }
1878:     }
1879:     */

```

```

1878: ZVAL_NULL(&retval);
1879: return FAILURE;
1880: }
1881: /* }}} */
1882:
1883: /* {{{ declare method parameters */
1884: /* supply a name and default to call by parameter */
1885: ZEND_BEGIN_ARG_INFO(arginfo_info_construct, 0)
1886:     ZEND_ARG_INFO(0, file_name)
1887: ZEND_END_ARG_INFO()
1888:
1889: ZEND_BEGIN_ARG_INFO_KK(arginfo_info_openFile, 0, 0, 0)
1890:     ZEND_ARG_INFO(0, open_mode)
1891:     ZEND_ARG_INFO(0, use_include_path)
1892:     ZEND_ARG_INFO(0, context)
1893: ZEND_END_ARG_INFO()
1894:
1895: ZEND_BEGIN_ARG_INFO_KK(arginfo_info_optimalFileInfoClass, 0, 0, 0)
1896:     ZEND_ARG_INFO(0, class_name)
1897: ZEND_END_ARG_INFO()
1898:
1899: ZEND_BEGIN_ARG_INFO_KK(arginfo_info_optimalSuffix, 0, 0, 0)
1900:     ZEND_ARG_INFO(0, suffix)
1901: ZEND_END_ARG_INFO()
1902:
1903: ZEND_BEGIN_ARG_INFO(arginfo_splFileInfo_void, 0)
1904: ZEND_END_ARG_INFO()
1905:
1906: /* the method table */
1907: /* each method can have its own parameters and visibility */
1908: static const zend_function_entry splFileInfo_functions[] = {
1909:     SPL_ME(splFileInfo, __construct, arginfo_info_construct, ZEND_ACC_PUBLIC)
1910:     SPL_ME(splFileInfo, getPath, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1911:     SPL_ME(splFileInfo, getFilename, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1912:     SPL_ME(splFileInfo, getExtension, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1913:     SPL_ME(splFileInfo, getBasename, arginfo_optimalSuffix, ZEND_ACC_PUBLIC)
1914:     SPL_ME(splFileInfo, getPathname, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1915:     SPL_ME(splFileInfo, getPerms, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1916:     SPL_ME(splFileInfo, getInode, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1917:     SPL_ME(splFileInfo, getSize, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1918:     SPL_ME(splFileInfo, getOwner, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1919:     SPL_ME(splFileInfo, getGroup, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1920:     SPL_ME(splFileInfo, getMTime, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1921:     SPL_ME(splFileInfo, getCTime, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1922:     SPL_ME(splFileInfo, getTyp, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1923:     SPL_ME(splFileInfo, isWritable, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1924:     SPL_ME(splFileInfo, isExecutable, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1925:     SPL_ME(splFileInfo, isReadable, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1926:     SPL_ME(splFileInfo, isExecutable, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1927:     SPL_ME(splFileInfo, isLink, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1928:     SPL_ME(splFileInfo, isDir, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1929:     SPL_ME(splFileInfo, isLink, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1930:     SPL_ME(splFileInfo, getLinkTarget, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1931:     IF HAVE_REALPATH 1 defined(OT)
1932:     SPL_ME(splFileInfo, getRealPath, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1933: #endif
1934:     SPL_ME(splFileInfo, getFileInfo, arginfo_info_optimalFileInfoClass, ZEND_ACC_PUBLIC)
1935:     SPL_ME(splFileInfo, getPathInfo, arginfo_info_optimalFileInfoClass, ZEND_ACC_PUBLIC)
1936:     SPL_ME(splFileInfo, openFile, arginfo_info_openFile, ZEND_ACC_PUBLIC)
1937:     SPL_ME(splFileInfo, setFileInfoClass, arginfo_info_optimalFileInfoClass, ZEND_ACC_PUBLIC)
1938:     SPL_ME(splFileInfo, setFileInfoClass, arginfo_info_optimalFileInfoClass, ZEND_ACC_PUBLIC)
1939:     SPL_ME(splFileInfo, __setState_W, NULL, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1940:     SPL_ME(splFileInfo, __toString, splFileInfo, getFilename, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1941:     PHP_FE_END
1942: };
1943:
1944: ZEND_BEGIN_ARG_INFO(arginfo_dir_construct, 0)
1945:     ZEND_ARG_INFO(0, path)
1946: ZEND_END_ARG_INFO()
1947:
1948: ZEND_BEGIN_ARG_INFO(arginfo_dir_iter_seek, 0)
1949:     ZEND_ARG_INFO(0, position)
1950: ZEND_END_ARG_INFO()
1951:
1952: /* the method table */
1953: /* each method can have its own parameters and visibility */
1954: static const zend_function_entry splDirectoryIterator_functions[] = {
1955:     SPL_ME(DirectoryIterator, __construct, arginfo_dir_construct, ZEND_ACC_PUBLIC)
1956:     SPL_ME(DirectoryIterator, getFilename, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1957:     SPL_ME(DirectoryIterator, getExtension, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1958:     SPL_ME(DirectoryIterator, getBasename, arginfo_optimalSuffix, ZEND_ACC_PUBLIC)
1959:     SPL_ME(DirectoryIterator, getPath, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1960:     SPL_ME(DirectoryIterator, rewind, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1961:     SPL_ME(DirectoryIterator, valid, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1962:     SPL_ME(DirectoryIterator, key, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1963:     SPL_ME(DirectoryIterator, current, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1964:     SPL_ME(DirectoryIterator, next, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1965:     SPL_ME(DirectoryIterator, next, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1966:     SPL_ME(DirectoryIterator, __toString, DirectoryIterator, getFilename, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1967:     PHP_FE_END
1968: };
1969:
1970: ZEND_BEGIN_ARG_INFO_KK(arginfo_dir_construct, 0, 0, 1)
1971:     ZEND_ARG_INFO(0, path)
1972:     ZEND_ARG_INFO(0, flags)
1973: ZEND_END_ARG_INFO()
1974:
1975: ZEND_BEGIN_ARG_INFO_KK(arginfo_dir_hasChildren, 0, 0, 0)
1976:     ZEND_ARG_INFO(0, allow_links)
1977: ZEND_END_ARG_INFO()
1978:
1979: ZEND_BEGIN_ARG_INFO_KK(arginfo_dir_setFlags, 0, 0, 0)
1980:     ZEND_ARG_INFO(0, flags)
1981: ZEND_END_ARG_INFO()
1982:
1983: static const zend_function_entry splFilesystemIterator_functions[] = {
1984:     SPL_ME(FilesystemIterator, __construct, arginfo_dir_construct, ZEND_ACC_PUBLIC)
1985:     SPL_ME(FilesystemIterator, rewind, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1986:     SPL_ME(FilesystemIterator, next, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1987:     SPL_ME(FilesystemIterator, key, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1988:     SPL_ME(FilesystemIterator, current, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1989:     SPL_ME(FilesystemIterator, getFlags, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1990:     SPL_ME(FilesystemIterator, setFlags, arginfo_dir_setFlags, ZEND_ACC_PUBLIC)
1991:     PHP_FE_END
1992: };
1993:
1994: static const zend_function_entry splRecursiveDirectoryIterator_functions[] = {
1995:     SPL_ME(RecursiveDirectoryIterator, __construct, arginfo_dir_construct, ZEND_ACC_PUBLIC)
1996:     SPL_ME(RecursiveDirectoryIterator, hasChildren, arginfo_dir_hasChildren, ZEND_ACC_PUBLIC)
1997:     SPL_ME(RecursiveDirectoryIterator, getChildren, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1998:     SPL_ME(RecursiveDirectoryIterator, getSubPath, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
1999:     SPL_ME(RecursiveDirectoryIterator, getSubPathname, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
2000:     PHP_FE_END
2001: };
2002:
2003: #if HAVE_GLOB
2004: static const zend_function_entry splGlobIterator_functions[] = {
2005:     SPL_ME(GlobIterator, __construct, arginfo_dir_construct, ZEND_ACC_PUBLIC)
2006:     SPL_ME(GlobIterator, count, arginfo_splFileInfo_void, ZEND_ACC_PUBLIC)
2007:     PHP_FE_END
2008: };
2009: #endif
2010:
2011: static int spl_filesystem_file_read(spl_filesystem_object *intern, int silent) /* {{{ */
2012: {
2013:     char *buf;
2014:     size_t line_len = 0;
2015:     zend_long line_add = (intern->u.file.current_line || IS_UNDEF(intern->u.file.current_val)) ? 1 : 0;
2016:     spl_filesystem_file_free_line(intern);
2017:
2018:     if (php_stream_eof(intern->u.file.stream)) {
2019:         if (!silent) {
2020:             zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Cannot read from file %s", intern->file_name);
2021:         }
2022:         return FAILURE;
2023:     }
2024:
2025:     if (intern->u.file.max_line_len > 0) {
2026:         buf = safe_emalloc(intern->u.file.max_line_len + 1, sizeof(char), 0);
2027:         if (php_stream_get_line(intern->u.file.stream, buf, intern->u.file.max_line_len + 1, sizeof(char)) == NULL) {
2028:             if (buf) {
2029:                 free(buf);
2030:             }
2031:             buf = NULL;
2032:         } else {
2033:             buf[line_len] = '\0';
2034:         }
2035:     } else {
2036:         buf = php_stream_get_line(intern->u.file.stream, NULL, 0, sizeof(char));
2037:     }
2038:
2039:     if (!buf) {
2040:         intern->u.file.current_line = estrdup("");
2041:         intern->u.file.current_line_len = 0;
2042:     } else {
2043:         if (SPL_BAS_FLAG(intern->flags, SPL_FILE_OBJECT_DROP_NEW_LINE)) {
2044:             line_len = strlen(buf, "\n");
2045:             buf[line_len] = '\0';
2046:         }
2047:
2048:         intern->u.file.current_line = buf;
2049:         intern->u.file.current_line_len = line_len;
2050:     }
2051:     intern->u.file.current_line_num += line_add;
2052:
2053:     return SUCCESS;
2054: } /* }}} */
2055:
2056: static int spl_filesystem_file_call(spl_filesystem_object *intern, zend_function *func_ptr, int pass_num_args, zval *return_value, zval *args) /* {{{ */
2057: {
2058:     zend_fcall_info fci;
2059:     zend_fcall_info_cache fci_cache;
2060:     zval *resource_ptr = intern->u.file.resource, *retval;
2061:     int result;
2062:     int num_args = pass_num_args + (args ? 2 : 1);
2063:     zval *params = (zval *)safe_emalloc(num_args, sizeof(zval), 0);
2064:
2065:     params[0] = *resource_ptr;
2066:
2067:     if (args) {
2068:         if (args[1] == *args;
2069:         params[1] = *args;
2070:         }
2071:
2072:     if (zend_get_parameters_array_ex(pass_num_args, params + (args ? 2 : 1)) != SUCCESS) {
2073:         efree(params);
2074:         WRONG_PARAM_COUNT_WITH_RETURN(FAILURE);
2075:     }
2076:
2077:     ZVAL_UNDEF(&retval);
2078:
2079:     fci.size = sizeof(fci);
2080:     fci.objekt = NULL;
2081:     fci.retval = &retval;
2082:     fci.param_count = num_args;
2083:     fci.params = params;
2084:     fci.no_separation = 1;
2085:     ZVAL_STR(&fci.function_name, func_ptr->common.function_name);
2086:
2087:     fci.function_handler = func_ptr;
2088:     fci.calling_scope = NULL;
2089:     fci.called_scope = NULL;
2090:     fci.object = NULL;
2091:
2092:     result = zend_call_function(&fci, &fci_cache);
2093:
2094:     if (result == FAILURE || IS_UNDEF(retval)) {
2095:         RETURN_FALSE;
2096:     } else {
2097:         ZVAL_ZVAL(&retval, result, 0, 0);
2098:     }
2099:
2100:     efree(params);
2101:     return result;
2102: } /* }}} */
2103:
2104: #define FileFunctionCall(func_name, pass_num_args, args) /* {{{ */
2105: {
2106:     zend_function *func_ptr;
2107:     func_ptr = zend_hash_str_find_ptr(CG(function_table), &func_name, sizeof(&func_name) - 1);
2108:     if (!func_ptr) {
2109:         zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Internal error, function '%s' not found. Please report", &func_name);
2110:     }
2111:     return func_ptr;
2112: }
2113:
2114: static int spl_filesystem_file_read_csv(spl_filesystem_object *intern, char delimiter, char enclosure, char escape, zval *return_value) /* {{{ */
2115: {
2116:     int ret = SUCCESS;
2117:     zval *value;
2118:
2119:     do {
2120:         spl_filesystem_file_read(intern, 1);
2121:         while (ret == SUCCESS & intern->u.file.current_line_len & SPL_BAS_FLAG(intern->flags, SPL_FILE_OBJECT_SKIP_EMPTY));
2122:
2123:         if (ret == SUCCESS) {
2124:             size_t buf_len = intern->u.file.current_line_len;
2125:             char *buf = estrdup(intern->u.file.current_line, buf_len);
2126:
2127:             if (!IS_UNDEF(intern->u.file.current_val)) {
2128:                 zval_ptr_stor(intern->u.file.current_val);
2129:                 ZVAL_UNDEF(intern->u.file.current_val);
2130:             }
2131:
2132:             php_getcsv(intern->u.file.stream, delimiter, enclosure, escape, buf_len, buf, intern->u.file.current_val);
2133:             zval_ptr_stor(intern->u.file.current_val);
2134:             value = &intern->u.file.current_val;
2135:             ZVAL_COPY(&value, value);
2136:             ZVAL_COPY(&return_value, value);
2137:         }
2138:     } while (ret == SUCCESS);
2139:     return ret;
2140: }
2141:
2142: static int spl_filesystem_file_read_line_ex(spl_filesystem_object *intern, int silent) /* {{{ */
2143: {
2144:     /* 1) use fgets(2) overlaid call the function, 3) do it directly */
2145:     if (SPL_BAS_FLAG(intern->flags, SPL_FILE_OBJECT_READ_CSV) || intern->u.file.func_ptr == &common_scope || spl_ce_SplFileObject {
2146:         if (php_stream_eof(intern->u.file.stream)) {
2147:             if (!silent) {
2148:                 zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Cannot read from file %s", intern->file_name);
2149:             }
2150:             return FAILURE;
2151:         }
2152:         if (SPL_BAS_FLAG(intern->flags, SPL_FILE_OBJECT_READ_CSV)) {
2153:             return spl_filesystem_file_read_csv(intern, intern->u.file.delimiter, intern->u.file.enclosure, intern->u.file.escape, NULL);
2154:         } else {
2155:             zend_execute_data *execute_data = EG(current_execute_data);
2156:             zend_call_method_with_0_params(this_ptr, SPL_OBJECT_EX(This), intern->u.file.func_ptr, "getCurrentLine", &retval);
2157:             if (!IS_UNDEF(retval)) {
2158:                 if (intern->u.file.current_line || IS_UNDEF(intern->u.file.current_val)) {
2159:                     intern->u.file.current_line_num++;
2160:                 }
2161:                 spl_filesystem_file_free_line(intern);
2162:                 if (!IS_STRING(retval)) {
2163:                     intern->u.file.current_line = estrdup((IS_STRING(retval) ? 2 : STRLEN(retval)));
2164:                 } else {
2165:                     zval *value = &retval;
2166:                     ZVAL_COPY(&value, value);
2167:                     ZVAL_COPY(&intern->u.file.current_val, value);
2168:                     zval_ptr_stor(&retval);
2169:                     return SUCCESS;
2170:                 }
2171:             } else {
2172:                 return FAILURE;
2173:             }
2174:         }
2175:     } else {
2176:         zval_ptr_stor(&retval);
2177:         return SUCCESS;
2178:     }
2179: }
2180:
2181: static int spl_filesystem_file_is_empty_line(spl_filesystem_object *intern) /* {{{ */
2182: {
2183:     if (intern->u.file.current_line) {
2184:         return intern->u.file.current_line_len == 0;
2185:     } else if (!IS_UNDEF(intern->u.file.current_val)) {
2186:         switch (IS_STRING(intern->u.file.current_val)) {
2187:             case IS_STRING:
2188:                 return 2 * STRLEN(intern->u.file.current_val) == 0;
2189:             case IS_ARRAY:
2190:                 if (SPL_BAS_FLAG(intern->flags, SPL_FILE_OBJECT_READ_CSV) & zend_hash_num_elements(Z_ARRVAL(intern->u.file.current_val)) == 1) {
2191:                     return 1;
2192:                 }
2193:                 return 2 * STRLEN(intern->u.file.current_val) == 0;
2194:             case IS_NULL:
2195:                 return 1;
2196:             default:
2197:                 return 0;
2198:         }
2199:     } else {
2200:         return 1;
2201:     }
2202: }
2203:
2204: static void spl_filesystem_file_read_line(spl_filesystem_object *intern, int silent) /* {{{ */
2205: {
2206:     while (SPL_BAS_FLAG(intern->flags, SPL_FILE_OBJECT_SKIP_EMPTY) & ret == SUCCESS & spl_filesystem_file_is_empty_line(intern)) {
2207:         spl_filesystem_file_free_line(intern);
2208:         ret = spl_filesystem_file_read_line_ex(this_ptr, intern, silent);
2209:     }
2210:     return ret;
2211: }
2212:
2213: static void spl_filesystem_file_rewind(spl_filesystem_object *intern) /* {{{ */
2214: {
2215:     if (!intern->u.file.stream) {
2216:         zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
2217:         return;
2218:     }
2219:     if (!php_stream_rewind(intern->u.file.stream)) {
2220:         zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Cannot rewind file %s", intern->file_name);
2221:     }
2222:     spl_filesystem_file_free_line(intern);
2223:     intern->u.file.current_line_num = 0;
2224:     if (SPL_BAS_FLAG(intern->flags, SPL_FILE_OBJECT_READ_READ)) {
2225:         spl_filesystem_file_read_line(this_ptr, intern, 1);
2226:     }
2227: }
2228:
2229: /* }}} */
2230:
2231: static void spl_filesystem_file_rewind(spl_filesystem_object *intern) /* {{{ */
2232: {
2233:     if (!intern->u.file.stream) {
2234:         zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
2235:         return;
2236:     }
2237:     if (!php_stream_rewind(intern->u.file.stream)) {
2238:         zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Cannot rewind file %s", intern->file_name);
2239:     }
2240:     spl_filesystem_file_free_line(intern);
2241:     intern->u.file.current_line_num = 0;
2242:     if (SPL_BAS_FLAG(intern->flags, SPL_FILE_OBJECT_READ_READ)) {
2243:         spl_filesystem_file_read_line(this_ptr, intern, 1);
2244:     }
2245: }
2246:
2247: /* }}} */
2248:
2249: /* {{{ proto void SplFileObject::__construct(string filename [, string mode = 'r'], bool use_include_path [, resource context]) */
2250: Construct a new file object
2251: SPL_METHOD(splFileObject, __construct)
2252: {

```

```

2254: spl_filesystem_object *intern = Z_SPFLIFILESYSTEM_P(getThis());
2255: zend_bool use_include_path = 0;
2256: char *pl, *p2;
2257: char *tmp_path;
2258: size_t tmp_path_len;
2259: zend_error_handling error_handling;
2260:
2261: intern->u.file.open_mode = NULL;
2262: intern->u.file.open_mode_len = 0;
2263:
2264: if (zend_parse_parameters_throw(ZEND_NUM_ARGS(), "p|abst",
2265:     &intern->file_name, &intern->file_name_len,
2266:     &intern->u.file.open_mode, &intern->u.file.open_mode_len,
2267:     &use_include_path, &intern->u.file.scontent) == FAILURE) {
2268:     intern->u.file.open_mode = NULL;
2269:     intern->file_name = NULL;
2270:     return;
2271: }
2272:
2273: if (intern->u.file.open_mode == NULL) {
2274:     intern->u.file.open_mode = "r";
2275:     intern->u.file.open_mode_len = 1;
2276: }
2277:
2278: zend_replace_error_handling(EH_THROW, spl_ce_RuntimeException, error_handling);
2279:
2280: if (spl_filesystem_file_open(intern, use_include_path, 0) == SUCCESS) {
2281:     tmp_path_len = strlen(intern->u.file.stream->orig_path);
2282:     if (tmp_path_len > 1 && IS_SLASH_AT(intern->u.file.stream->orig_path, tmp_path_len-1)) {
2283:         tmp_path_len--;
2284:     }
2285:     tmp_path = estrndup(intern->u.file.stream->orig_path, tmp_path_len);
2286:
2287:     pl = strchr(tmp_path, '/');
2288:     if (defined(PHP_WIN32))
2289:         p2 = strchr(tmp_path, '\\');
2290:     else
2291:         p2 = strchr(tmp_path, '\0');
2292:     if (pl || p2) {
2293:         intern->u.path_len = (pl > p2 ? pl : p2) - tmp_path;
2294:     } else {
2295:         intern->u.path_len = 0;
2296:     }
2297:     if (p2 == 0)
2298:         p2 = tmp_path;
2299:     if (pl || p2) {
2300:         if (pl > p2)
2301:             intern->u.path = estrndup(intern->u.file.stream->orig_path, intern->u.path_len);
2302:         else
2303:             intern->u.path = estrndup(intern->u.file.stream->orig_path, intern->u.path_len);
2304:     }
2305:     zend_restore_error_handling(error_handling);
2306: }
2307: /* {} */
2308:
2309: /* {{{ proto void SplTempFileObject::__construct([int max_memory])
2310: Construct a new temp file object */
2311: SPL_METHOD(SplTempFileObject, __construct)
2312: {
2313:     zend_long max_memory = PHP_STREAM_MAX_LEN;
2314:     char tmp_fname[40];
2315:     spl_filesystem_object *intern = Z_SPFLIFILESYSTEM_P(getThis());
2316:     zend_error_handling error_handling;
2317:
2318:     if (zend_parse_parameters_throw(ZEND_NUM_ARGS(), "|i", &max_memory) == FAILURE) {
2319:         return;
2320:     }
2321:
2322:     if (max_memory < 0) {
2323:         intern->file_name = "php://memory";
2324:         intern->file_name_len = 12;
2325:     } else if (ZEND_NUM_ARGS() == 1) {
2326:         intern->file_name_len = sprintf(tmp_fname, "%s%04d", "php://tmp/memmemory:", ZEND_LONG_FMT, max_memory);
2327:         intern->file_name = tmp_fname;
2328:     } else {
2329:         intern->file_name = "php://temp/";
2330:         intern->file_name_len = 10;
2331:     }
2332:     intern->u.file.open_mode = "wb";
2333:     intern->u.file.open_mode_len = 1;
2334:
2335:     zend_replace_error_handling(EH_THROW, spl_ce_RuntimeException, error_handling);
2336:     if (spl_filesystem_file_open(intern, 0, 0) == SUCCESS) {
2337:         intern->u.path_len = 0;
2338:         intern->u.path = estrdup("");
2339:     }
2340:     zend_restore_error_handling(error_handling);
2341: } /* {} */
2342:
2343: /* {{{ proto void SplFileObject::rewind()
2344: Rewind the file and read the first line */
2345: SPL_METHOD(SplFileObject, rewind)
2346: {
2347:     spl_filesystem_object *intern = Z_SPFLIFILESYSTEM_P(getThis());
2348:
2349:     if (zend_parse_parameters_none() == FAILURE) {
2350:         return;
2351:     }
2352:
2353:     spl_filesystem_file_rewind(getThis(), intern);
2354: } /* {} */
2355:
2356: /* {{{ proto void SplFileObject::eof()
2357: Return whether end of file is reached */
2358: SPL_METHOD(SplFileObject, eof)
2359: {
2360:     spl_filesystem_object *intern = Z_SPFLIFILESYSTEM_P(getThis());
2361:
2362:     if (zend_parse_parameters_none() == FAILURE) {
2363:         return;
2364:     }
2365:
2366:     if (intern->u.file.stream) {
2367:         zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialised");
2368:         return;
2369:     }
2370:
2371:     RETURN_BOOL(php_stream_eof(intern->u.file.stream));
2372: } /* {} */
2373:
2374: /* {{{ proto void SplFileObject::valid()
2375: Return fact() */
2376: SPL_METHOD(SplFileObject, valid)
2377: {
2378:     spl_filesystem_object *intern = Z_SPFLIFILESYSTEM_P(getThis());
2379:
2380:     if (zend_parse_parameters_none() == FAILURE) {
2381:         return;
2382:     }
2383:
2384:     if (SPL_HAS_FLAG(intern->flags, SPL_FILE_OBJECT_READ_AHEAD)) {
2385:         RETURN_BOOL(intern->u.file.current_line != 1 && !IS_EOFED(intern->u.file.current_val));
2386:     } else {
2387:         if (intern->u.file.stream) {
2388:             RETURN_FALSE;
2389:         }
2390:         RETURN_BOOL(php_stream_eof(intern->u.file.stream));
2391:     }
2392: } /* {} */
2393:
2394: /* {{{ proto string SplFileObject::fgets()
2395: Return next line from file */
2396: SPL_METHOD(SplFileObject, fgets)
2397: {
2398:     spl_filesystem_object *intern = Z_SPFLIFILESYSTEM_P(getThis());
2399:
2400:     if (zend_parse_parameters_none() == FAILURE) {
2401:         return;
2402:     }
2403:
2404:     if (intern->u.file.stream) {
2405:         zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialised");
2406:         return;
2407:     }
2408:     if (spl_filesystem_file_read(intern, 0) == FAILURE) {
2409:         RETURN_FALSE;
2410:     }
2411:     RETURN_STRINGL(intern->u.file.current_line, intern->u.file.current_line_len);
2412: } /* {} */
2413:
2414: /* {{{ proto string SplFileObject::current()
2415: Return current line from file */
2416: SPL_METHOD(SplFileObject, current)
2417: {
2418:     spl_filesystem_object *intern = Z_SPFLIFILESYSTEM_P(getThis());
2419:
2420:     if (zend_parse_parameters_none() == FAILURE) {
2421:         return;
2422:     }
2423:
2424:     if (intern->u.file.stream) {
2425:         zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialised");
2426:         return;
2427:     }
2428:
2429:     if (intern->u.file.current_line & !IS_EOFED(intern->u.file.current_val)) {
2430:         RETURN_STRINGL(intern->u.file.current_line, intern->u.file.current_line_len);
2431:     } else if (!IS_EOFED(intern->u.file.current_val)) {
2432:         zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialised");
2433:         return;
2434:     }
2435:     RETURN_STRINGL(intern->u.file.current_line, intern->u.file.current_line_len);
2436: }
2437:
2438: ZVAL_DEREF(value);
2439: ZVAL_COPY(return_value, value);
2440: return;
2441: }
2442:
2443: /* {{{ proto void SplFileObject::next()
2444: Return next line from file */
2445: SPL_METHOD(SplFileObject, next)
2446: {
2447:     spl_filesystem_object *intern = Z_SPFLIFILESYSTEM_P(getThis());
2448:
2449:     if (zend_parse_parameters_none() == FAILURE) {
2450:         return;
2451:     }
2452:
2453:     if (intern->u.file.stream) {
2454:         zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialised");
2455:         return;
2456:     }
2457:     if (spl_filesystem_file_read(intern, 0) == FAILURE) {
2458:         RETURN_FALSE;
2459:     }
2460:     RETURN_STRINGL(intern->u.file.current_line, intern->u.file.current_line_len);
2461: }
2462:
2463: /* {{{ proto void SplFileObject::setFlags(int flags)
2464: Set file handling flags */
2465: SPL_METHOD(SplFileObject, setFlags)
2466: {
2467:     spl_filesystem_object *intern = Z_SPFLIFILESYSTEM_P(getThis());
2468:
2469:     if (zend_parse_parameters_none() == FAILURE) {
2470:         return;
2471:     }
2472:
2473:     if (SPL_HAS_FLAG(intern->flags, SPL_FILE_OBJECT_READ_AHEAD)) {
2474:         spl_filesystem_file_read_line(getThis(), intern, 1);
2475:     }
2476:     intern->u.file.current_line_num++;
2477: } /* {} */
2478:
2479: /* {{{ proto void SplFileObject::setFlags(int flags)
2480: Set file handling flags */
2481: SPL_METHOD(SplFileObject, setFlags)
2482: {
2483:     spl_filesystem_object *intern = Z_SPFLIFILESYSTEM_P(getThis());
2484:
2485:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "|i", &intern->flags) == FAILURE) {
2486:         return;
2487:     }
2488: } /* {} */
2489:
2490: /* {{{ proto int SplFileObject::getFlags()
2491: Get file handling flags */
2492: SPL_METHOD(SplFileObject, getFlags)
2493: {
2494:     spl_filesystem_object *intern = Z_SPFLIFILESYSTEM_P(getThis());
2495:
2496:     if (zend_parse_parameters_none() == FAILURE) {
2497:         return;
2498:     }
2499:
2500:     RETURN_LONG(intern->flags & SPL_FILE_OBJECT_MASK);
2501: } /* {} */
2502:
2503: /* {{{ proto void SplFileObject::setMaxLineLen(int max_len)
2504: Set maximum line length */
2505: SPL_METHOD(SplFileObject, setMaxLineLen)
2506: {
2507:     zend_long max_len;
2508:
2509:     spl_filesystem_object *intern = Z_SPFLIFILESYSTEM_P(getThis());
2510:
2511:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "|i", &max_len) == FAILURE) {
2512:         return;
2513:     }
2514:
2515:     if (max_len < 0) {
2516:         zend_throw_exception_ex(spl_ce_DomainException, 0, "Maximum line length must be greater than or equal zero");
2517:         return;
2518:     }
2519:
2520:     intern->u.file.max_line_len = max_len;
2521: } /* {} */
2522:
2523: /* {{{ proto int SplFileObject::getMaxLineLen()
2524: Get maximum line length */
2525: SPL_METHOD(SplFileObject, getMaxLineLen)
2526: {
2527:     spl_filesystem_object *intern = Z_SPFLIFILESYSTEM_P(getThis());
2528:
2529:     if (zend_parse_parameters_none() == FAILURE) {
2530:         return;
2531:     }
2532:
2533:     RETURN_LONG((zend_long)intern->u.file.max_line_len);
2534: } /* {} */
2535:
2536: /* {{{ proto bool SplFileObject::hasChildren()
2537: Return false */
2538: SPL_METHOD(SplFileObject, hasChildren)
2539: {
2540:     if (zend_parse_parameters_none() == FAILURE) {
2541:         return;

```

```

2629:     php_error_docref(NULL, E_WARNING, "escape must be a character");
2630:     RETURN_FALSE;
2631: }
2632: escape = esc[0];
2633: /* no break */
2634: case 3:
2635:     if (d_len != 1) {
2636:         php_error_docref(NULL, E_WARNING, "enclosure must be a character");
2637:         RETURN_FALSE;
2638:     }
2639:     enclosure = enclo[0];
2640:     /* no break */
2641: case 2:
2642:     if (d_len != 1) {
2643:         php_error_docref(NULL, E_WARNING, "delimiter must be a character");
2644:         RETURN_FALSE;
2645:     }
2646:     delimiter = delim[0];
2647:     /* no break */
2648: case 1:
2649: case 0:
2650:     /* break */
2651: }
2652: ret = php_fputcs(intern->u.file.stream, fields, delimiter, enclosure, escape);
2653: RETURN_LONG(ret);
2654: }
2655: }
2656: /* }}} */
2657:
2658: /* {{{ proto void SplFileObject::setCvsControl([string delimiter [, string enclosure [, string escape]])
2659:  * Get the delimiter, enclosure and escape character used in fgets */
2660: SPL_METHOD(SplFileObject, setCvsControl)
2661: {
2662:     spl_filesystem_object *intern = _SPL_FILESYSTEM_P(getThis());
2663:     char delimiter = '\n', enclosure = '"', escape = '\\';
2664:     char *delim = NULL, *enclo = NULL, *esc = NULL;
2665:     size_t d_len = 0, e_len = 0, esc_len = 0;
2666:
2667:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "[sss]", &delim, &d_len, &enclo, &e_len, &esc, &esc_len) == SUCCESS) {
2668:         switch(ZEND_NUM_ARGS()) {
2669:             case 3:
2670:                 if (d_len != 1) {
2671:                     php_error_docref(NULL, E_WARNING, "escape must be a character");
2672:                     RETURN_FALSE;
2673:                 }
2674:                 escape = esc[0];
2675:                 /* no break */
2676:             case 2:
2677:                 if (e_len != 1) {
2678:                     php_error_docref(NULL, E_WARNING, "enclosure must be a character");
2679:                     RETURN_FALSE;
2680:                 }
2681:                 enclosure = enclo[0];
2682:                 /* no break */
2683:             case 1:
2684:                 if (d_len != 1) {
2685:                     php_error_docref(NULL, E_WARNING, "delimiter must be a character");
2686:                     RETURN_FALSE;
2687:                 }
2688:                 delimiter = delim[0];
2689:                 /* no break */
2690:             case 0:
2691:                 /* break */
2692:             }
2693:         }
2694:         intern->u.file.delimiter = delimiter;
2695:         intern->u.file.enclosure = enclosure;
2696:         intern->u.file.escape = escape;
2697:     }
2698:     /* }}} */
2699:
2700: /* {{{ proto array SplFileObject::getCvsControl()
2701:  * Get the delimiter, enclosure and escape character used in fgets */
2702: SPL_METHOD(SplFileObject, getCvsControl)
2703: {
2704:     spl_filesystem_object *intern = _SPL_FILESYSTEM_P(getThis());
2705:     char delimiter[2], enclosure[2], escape[2];
2706:     array_init(return_value);
2707:
2708:     delimiter[0] = intern->u.file.delimiter;
2709:     delimiter[1] = '\0';
2710:     enclosure[0] = intern->u.file.enclosure;
2711:     enclosure[1] = '\0';
2712:     escape[0] = intern->u.file.escape;
2713:     escape[1] = '\0';
2714:
2715:     add_next_index_string(return_value, delimiter);
2716:     add_next_index_string(return_value, enclosure);
2717:     add_next_index_string(return_value, escape);
2718: }
2719: /* }}} */
2720:
2721: /* {{{ proto bool SplFileObject::flock(int operation [, int wouldblock])
2722:  * Portable file locking */
2723: SPL_FUNCTION(flock)
2724: {
2725:     /* }}} */
2726:
2727: /* {{{ proto bool SplFileObject::ftruncate()
2728:  * Flush the file */
2729: SPL_METHOD(SplFileObject, ftruncate)
2730: {
2731:     spl_filesystem_object *intern = _SPL_FILESYSTEM_P(getThis());
2732:
2733:     if(!intern->u.file.stream) {
2734:         zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
2735:         return;
2736:     }
2737:
2738:     RETURN_BOOL(php_stream_flush(intern->u.file.stream));
2739:     /* }}} */
2740:
2741: /* {{{ proto int SplFileObject::ftell()
2742:  * Return current file position */
2743: SPL_METHOD(SplFileObject, ftell)
2744: {
2745:     spl_filesystem_object *intern = _SPL_FILESYSTEM_P(getThis());
2746:     zend_long ret;
2747:
2748:     if(!intern->u.file.stream) {
2749:         zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
2750:         return;
2751:     }
2752:
2753:     ret = php_stream_tell(intern->u.file.stream);
2754:
2755:     if (ret == -1) {
2756:         RETURN_FALSE;
2757:     } else {
2758:         RETURN_LONG(ret);
2759:     }
2760: }
2761: /* }}} */
2762:
2763: /* {{{ proto int SplFileObject::fseek(int pos [, int whence = SEEK_SET])
2764:  * Return current file position */
2765: SPL_METHOD(SplFileObject, fseek)
2766: {
2767:     spl_filesystem_object *intern = _SPL_FILESYSTEM_P(getThis());
2768:     zend_long pos, whence = SEEK_SET;
2769:
2770:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "[i!]", &pos, &whence) == FAILURE) {
2771:         return;
2772:     }
2773:
2774:     if(!intern->u.file.stream) {
2775:         zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
2776:         return;
2777:     }
2778:
2779:     spl_filesystem_file_free_line(intern);
2780:     RETURN_LONG(php_stream_seek(intern->u.file.stream, pos, (int)whence));
2781:     /* }}} */
2782:
2783: /* {{{ proto int SplFileObject::fgetc()
2784:  * Get a character from the file */
2785: SPL_METHOD(SplFileObject, fgetc)
2786: {
2787:     spl_filesystem_object *intern = _SPL_FILESYSTEM_P(getThis());
2788:     char buf[2];
2789:     int result;
2790:
2791:     if(!intern->u.file.stream) {
2792:         zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
2793:         return;
2794:     }
2795:
2796:     spl_filesystem_file_free_line(intern);
2797:
2798:     result = php_stream_getc(intern->u.file.stream);
2799:
2800:     if (result == EOF) {
2801:         RETURN_FALSE;
2802:     } else {
2803:         if (result == '\n') {
2804:             intern->u.file.current_line_num++;
2805:         }
2806:         buf[0] = result;
2807:         buf[1] = '\0';
2808:
2809:         RETURN_STRING(buf, 1);
2810:     }
2811:     /* }}} */
2812:
2813: /* {{{ proto string SplFileObject::fgets([string allowable_tags])
2814:  * Get a line from file pointer and strip HTML tags */
2815: SPL_METHOD(SplFileObject, fgets)
2816:
2817:     spl_filesystem_object *intern = _SPL_FILESYSTEM_P(getThis());
2818:     zend_long ret;
2819:
2820:     if(!intern->u.file.stream) {
2821:         zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
2822:         return;
2823:     }
2824:
2825:     RETURN_LONG(php_stream_read(intern->u.file.stream, str, str_len));
2826:
2827:     if (length >= 0) {
2828:         str_len = MIN((size_t)length, str_len);
2829:     } else {
2830:         /* Negative length given, nothing to write */
2831:         str_len = 0;
2832:     }
2833:
2834:     if (!str_len) {
2835:         RETURN_LONG(0);
2836:     }
2837:
2838:     RETURN_LONG(php_stream_write(intern->u.file.stream, str, str_len));
2839:     /* }}} */
2840:
2841: SPL_METHOD(SplFileObject, fread)
2842: {
2843:     spl_filesystem_object *intern = _SPL_FILESYSTEM_P(getThis());
2844:     zend_long length = 0;
2845:
2846:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "[i]", &str, &str_len, &length) == FAILURE) {
2847:         return;
2848:     }
2849:
2850:     if(!intern->u.file.stream) {
2851:         zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
2852:         return;
2853:     }
2854:
2855:     if (ZEND_NUM_ARGS() > 1) {
2856:         if (length >= 0) {
2857:             str_len = MIN((size_t)length, str_len);
2858:         } else {
2859:             /* Negative length given, nothing to write */
2860:             str_len = 0;
2861:         }
2862:     }
2863:
2864:     if (!str_len) {
2865:         RETURN_LONG(0);
2866:     }
2867:
2868:     RETURN_LONG(php_stream_write(intern->u.file.stream, str, str_len));
2869:     /* }}} */
2870:
2871: SPL_METHOD(SplFileObject, fread)
2872: {
2873:     spl_filesystem_object *intern = _SPL_FILESYSTEM_P(getThis());
2874:     zend_long length = 0;
2875:
2876:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "[i]", &length) == FAILURE) {
2877:         return;
2878:     }
2879:
2880:     if(!intern->u.file.stream) {
2881:         zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
2882:         return;
2883:     }
2884:
2885:     if (length <= 0) {
2886:         php_error_docref(NULL, E_WARNING, "Length parameter must be greater than 0");
2887:         RETURN_FALSE;
2888:     }
2889:
2890:     if (length >= 0) {
2891:         zend_string_alloc(length, 0);
2892:         Z_STRLEN_P(return_value) = php_stream_read(intern->u.file.stream, Z_STRVAL_P(return_value), length);
2893:     }
2894:
2895:     /* needed because read/readgz doesn't put a null at the end */
2896:     Z_STRVAL_P(return_value)[Z_STRLEN_P(return_value)] = 0;
2897:
2898:     /* {{{ proto bool SplFileObject::fstat()
2899:     * Stat() on a filehandle */
2900:     SPL_FUNCTION(fstat)
2901:     {
2902:         /* }}} */
2903:
2904:     /* {{{ proto bool SplFileObject::truncate(int size)
2905:     * Truncate file to 'size' length */
2906:     SPL_METHOD(SplFileObject, truncate)
2907:     {
2908:         spl_filesystem_object *intern = _SPL_FILESYSTEM_P(getThis());
2909:         zend_long size;
2910:
2911:         if (zend_parse_parameters(ZEND_NUM_ARGS(), "[i]", &size) == FAILURE) {
2912:             return;
2913:         }
2914:
2915:         if(!intern->u.file.stream) {
2916:             zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
2917:             return;
2918:         }
2919:
2920:         if (php_stream_truncate_supported(intern->u.file.stream) {
2921:             zend_throw_exception_ex(spl_ce_LogicException, 0, "Can't truncate file to", intern->u.file.name);
2922:             RETURN_FALSE;
2923:         }
2924:
2925:         RETURN_BOOL(0 == php_stream_truncate_set_size(intern->u.file.stream, size));
2926:     }
2927:     /* }}} */
2928:
2929: /* {{{ proto void SplFileObject::seek(int line_pos)
2930:  * Seek to specified line */
2931: SPL_METHOD(SplFileObject, seek)
2932: {
2933:     spl_filesystem_object *intern = _SPL_FILESYSTEM_P(getThis());
2934:     zend_long line_pos;
2935:
2936:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "[i]", &line_pos) == FAILURE) {
2937:         return;
2938:     }
2939:
2940:     if(!intern->u.file.stream) {
2941:         zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
2942:         return;
2943:     }
2944:
2945:     if (line_pos < 0) {
2946:         zend_throw_exception_ex(spl_ce_LogicException, 0, "Can't seek file to negative line", zend_long_PWT, intern->u.file.name, line_pos);
2947:         RETURN_FALSE;
2948:     }
2949:
2950:     spl_filesystem_file_rewind(getThis(), intern);
2951:
2952:     while(intern->u.file.current_line_num < line_pos) {
2953:         if (spl_filesystem_file_read_line(getThis()) == FAILURE) {
2954:             break;
2955:         }
2956:     }
2957:     /* }}} */
2958:
2959: /* {{{ Function/Class/Method definitions */
2960: ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_construct, 0, 1)
2961:     ZEND_ARG_INFO(0, file_name)
2962:     ZEND_ARG_INFO(0, open_mode)
2963:     ZEND_ARG_INFO(0, use_include_path)
2964:     ZEND_ARG_INFO(0, context)
2965: ZEND_END_ARG_INFO()
2966:
2967: ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_setFlags, 0)
2968:     ZEND_ARG_INFO(0, flags)
2969: ZEND_END_ARG_INFO()
2970:
2971: ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_setMaxLineLen, 0)
2972:     ZEND_ARG_INFO(0, max_len)
2973: ZEND_END_ARG_INFO()
2974:
2975: ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_fgetc, 0, 0)
2976:     ZEND_ARG_INFO(0, delimiter)
2977:     ZEND_ARG_INFO(0, enclosure)
2978:     ZEND_ARG_INFO(0, escape)
2979: ZEND_END_ARG_INFO()
2980:
2981: ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_fgets, 0, 0)
2982:     ZEND_ARG_INFO(0, delimiter)
2983:     ZEND_ARG_INFO(0, enclosure)
2984:     ZEND_ARG_INFO(0, escape)
2985:     ZEND_ARG_INFO(0, allowable_tags)
2986: ZEND_END_ARG_INFO()
2987:
2988: ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_fread, 0, 0)
2989:     ZEND_ARG_INFO(0, length)
2990:     ZEND_ARG_INFO(0, str)
2991:     ZEND_ARG_INFO(0, str_len)
2992:     ZEND_ARG_INFO(0, length)
2993: ZEND_END_ARG_INFO()
2994:
2995: ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_ftruncate, 0, 0)
2996:     ZEND_ARG_INFO(0, size)
2997: ZEND_END_ARG_INFO()
2998:
2999: ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_seek, 0, 0)
3000:     ZEND_ARG_INFO(0, line_pos)
3001: ZEND_END_ARG_INFO()
3002:
3003: ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_stat, 0, 0)
3004:     ZEND_ARG_INFO(0, file_name)
3005:     ZEND_ARG_INFO(0, open_mode)
3006:     ZEND_ARG_INFO(0, use_include_path)
3007:     ZEND_ARG_INFO(0, context)
3008:     ZEND_ARG_INFO(0, file_name)
3009:     ZEND_ARG_INFO(0, open_mode)
3010:     ZEND_ARG_INFO(0, use_include_path)
3011:     ZEND_ARG_INFO(0, context)
3012:     ZEND_ARG_INFO(0, file_name)
3013:     ZEND_ARG_INFO(0, open_mode)
3014:     ZEND_ARG_INFO(0, use_include_path)
3015:     ZEND_ARG_INFO(0, context)
3016:     ZEND_ARG_INFO(0, file_name)
3017:     ZEND_ARG_INFO(0, open_mode)
3018:     ZEND_ARG_INFO(0, use_include_path)
3019:     ZEND_ARG_INFO(0, context)
3020:     ZEND_ARG_INFO(0, file_name)
3021:     ZEND_ARG_INFO(0, open_mode)
3022:     ZEND_ARG_INFO(0, use_include_path)
3023:     ZEND_ARG_INFO(0, context)
3024:     ZEND_ARG_INFO(0, file_name)
3025:     ZEND_ARG_INFO(0, open_mode)
3026:     ZEND_ARG_INFO(0, use_include_path)
3027:     ZEND_ARG_INFO(0, context)
3028:     ZEND_ARG_INFO(0, file_name)
3029:     ZEND_ARG_INFO(0, open_mode)
3030:     ZEND_ARG_INFO(0, use_include_path)
3031:     ZEND_ARG_INFO(0, context)
3032:     ZEND_ARG_INFO(0, file_name)
3033:     ZEND_ARG_INFO(0, open_mode)
3034:     ZEND_ARG_INFO(0, use_include_path)
3035:     ZEND_ARG_INFO(0, context)
3036:     ZEND_ARG_INFO(0, file_name)
3037:     ZEND_ARG_INFO(0, open_mode)
3038:     ZEND_ARG_INFO(0, use_include_path)
3039:     ZEND_ARG_INFO(0, context)
3040:     ZEND_ARG_INFO(0, file_name)
3041:     ZEND_ARG_INFO(0, open_mode)
3042:     ZEND_ARG_INFO(0, use_include_path)
3043:     ZEND_ARG_INFO(0, context)
3044:     ZEND_ARG_INFO(0, file_name)
3045:     ZEND_ARG_INFO(0, open_mode)
3046:     ZEND_ARG_INFO(0, use_include_path)
3047:     ZEND_ARG_INFO(0, context)
3048:     ZEND_ARG_INFO(0, file_name)
3049:     ZEND_ARG_INFO(0, open_mode)
3050:     ZEND_ARG_INFO(0, use_include_path)
3051:     ZEND_ARG_INFO(0, context)
3052:     ZEND_ARG_INFO(0, file_name)
3053:     ZEND_ARG_INFO(0, open_mode)
3054:     ZEND_ARG_INFO(0, use_include_path)
3055:     ZEND_ARG_INFO(0, context)
3056:     ZEND_ARG_INFO(0, file_name)
3057:     ZEND_ARG_INFO(0, open_mode)
3058:     ZEND_ARG_INFO(0, use_include_path)
3059:     ZEND_ARG_INFO(0, context)
3060:     ZEND_ARG_INFO(0, file_name)
3061:     ZEND_ARG_INFO(0, open_mode)
3062:     ZEND_ARG_INFO(0, use_include_path)
3063:     ZEND_ARG_INFO(0, context)
3064:     ZEND_ARG_INFO(0, file_name)
3065:     ZEND_ARG_INFO(0, open_mode)
3066:     ZEND_ARG_INFO(0, use_include_path)
3067:     ZEND_ARG_INFO(0, context)
3068:     ZEND_ARG_INFO(0, file_name)
3069:     ZEND_ARG_INFO(0, open_mode)
3070:     ZEND_ARG_INFO(0, use_include_path)
3071:     ZEND_ARG_INFO(0, context)
3072:     ZEND_ARG_INFO(0, file_name)
3073:     ZEND_ARG_INFO(0, open_mode)
3074:     ZEND_ARG_INFO(0, use_include_path)
3075:     ZEND_ARG_INFO(0, context)
3076:     ZEND_ARG_INFO(0, file_name)
3077:     ZEND_ARG_INFO(0, open_mode)
3078:     ZEND_ARG_INFO(0, use_include_path)
3079:     ZEND_ARG_INFO(0, context)
3080:     ZEND_ARG_INFO(0, file_name)
3081:     ZEND_ARG_INFO(0, open_mode)
3082:     ZEND_ARG_INFO(0, use_include_path)
3083:     ZEND_ARG_INFO(0, context)
3084:     ZEND_ARG_INFO(0, file_name)
3085:     ZEND_ARG_INFO(0, open_mode)
3086:     ZEND_ARG_INFO(0, use_include_path)
3087:     ZEND_ARG_INFO(0, context)
3088:     ZEND_ARG_INFO(0, file_name)
3089:     ZEND_ARG_INFO(0, open_mode)
3090:     ZEND_ARG_INFO(0, use_include_path)
3091:     ZEND_ARG_INFO(0, context)
3092:     ZEND_ARG_INFO(0, file_name)
3093:     ZEND_ARG_INFO(0, open_mode)
3094:     ZEND_ARG_INFO(0, use_include_path)
3095:     ZEND_ARG_INFO(0, context)
3096:     ZEND_ARG_INFO(0, file_name)
3097:     ZEND_ARG_INFO(0, open_mode)
3098:     ZEND_ARG_INFO(0, use_include_path)
3099:     ZEND_ARG_INFO(0, context)
3100:     ZEND_ARG_INFO(0, file_name)
3101:     ZEND_ARG_INFO(0, open_mode)
3102:     ZEND_ARG_INFO(0, use_include_path)
3103:     ZEND_ARG_INFO(0, context)
3104:     ZEND_ARG_INFO(0, file_name)
3105:     ZEND_ARG_INFO(0, open_mode)
3106:     ZEND_ARG_INFO(0, use_include_path)
3107:     ZEND_ARG_INFO(0, context)
3108:     ZEND_ARG_INFO(0, file_name)
3109:     ZEND_ARG_INFO(0, open_mode)
3110:     ZEND_ARG_INFO(0, use_include_path)
3111:     ZEND_ARG_INFO(0, context)
3112:     ZEND_ARG_INFO(0, file_name)
3113:     ZEND_ARG_INFO(0, open_mode)
3114:     ZEND_ARG_INFO(0, use_include_path)
3115:     ZEND_ARG_INFO(0, context)
3116:     ZEND_ARG_INFO(0, file_name)
3117:     ZEND_ARG_INFO(0, open_mode)
3118:     ZEND_ARG_INFO(0, use_include_path)
3119:     ZEND_ARG_INFO(0, context)
3120:     ZEND_ARG_INFO(0, file_name)
3121:     ZEND_ARG_INFO(0, open_mode)
3122:     ZEND_ARG_INFO(0, use_include_path)
3123:     ZEND_ARG_INFO(0, context)
3124:     ZEND_ARG_INFO(0, file_name)
3125:     ZEND_ARG_INFO(0, open_mode)
3126:     ZEND_ARG_INFO(0, use_include_path)
3127:     ZEND_ARG_INFO(0, context)
3128:     ZEND_ARG_INFO(0, file_name)
3129:     ZEND_ARG_INFO(0, open_mode)
3130:     ZEND_ARG_INFO(0, use_include_path)
3131:     ZEND_ARG_INFO(0, context)
3132:     ZEND_ARG_INFO(0, file_name)
3133:     ZEND_ARG_INFO(0, open_mode)
3134:     ZEND_ARG_INFO(0, use_include_path)
3135:     ZEND_ARG_INFO(0, context)
3136:     ZEND_ARG_INFO(0, file_name)
3137:     ZEND_ARG_INFO(0, open_mode)
3138:     ZEND_ARG_INFO(0, use_include_path)
3139:     ZEND_ARG_INFO(0, context)
3140:     ZEND_ARG_INFO(0, file_name)
3141:     ZEND_ARG_INFO(0, open_mode)
3142:     ZEND_ARG_INFO(0, use_include_path)
3143:     ZEND_ARG_INFO(0, context)
3144:     ZEND_ARG_INFO(0, file_name)
3145:     ZEND_ARG_INFO(0, open_mode)
3146:     ZEND_ARG_INFO(0, use_include_path)
3147:     ZEND_ARG_INFO(0, context)
3148:     ZEND_ARG_INFO(0, file_name)
3149:     ZEND_ARG_INFO(0, open_mode)
3150:     ZEND_ARG_INFO(0, use_include_path)
3151:     ZEND_ARG_INFO(0, context)
3152:     ZEND_ARG_INFO(0, file_name)
3153:     ZEND_ARG_INFO(0, open_mode)
3154:     ZEND_ARG_INFO(0, use_include_path)
3155:     ZEND_ARG_INFO(0, context)
3156:     ZEND_ARG_INFO(0, file_name)
3157:     ZEND_ARG_INFO(0, open_mode)
3158:     ZEND_ARG_INFO(0, use_include_path)
3159:     ZEND_ARG_INFO(0, context)
3160:     ZEND_ARG_INFO(0, file_name)
3161:     ZEND_ARG_INFO(0, open_mode)
3162:     ZEND_ARG_INFO(0, use_include_path)
3163:     ZEND_ARG_INFO(0, context)
3164:     ZEND_ARG_INFO(0, file_name)
3165:     ZEND_ARG_INFO(0, open_mode)
3166:     ZEND_ARG_INFO(0, use_include_path)
3167:     ZEND_ARG_INFO(0, context)
3168:     ZEND_ARG_INFO(0, file_name)
3169:     ZEND_ARG_INFO(0, open_mode)
3170:     ZEND_ARG_INFO(0, use_include_path)
3171:     ZEND_ARG_INFO(0, context)
3172:     ZEND_ARG_INFO(0, file_name)
3173:     ZEND_ARG_INFO(0, open_mode)
3174:     ZEND_ARG_INFO(0, use_include_path)
3175:     ZEND_ARG_INFO(0, context)
3176:     ZEND_ARG_INFO(0, file_name)
3177:     ZEND_ARG_INFO(0, open_mode)
3178:     ZEND_ARG_INFO(0, use_include_path)
3179:     ZEND_ARG_INFO(0, context)
3180:     ZEND_ARG_INFO(0, file_name)
3181:     ZEND_ARG_INFO(0, open_mode)
3182:     ZEND_ARG_INFO(0, use_include_path)
3183:     ZEND_ARG_INFO(0, context)
3184:     ZEND_ARG_INFO(0, file_name)
3185:     ZEND_ARG_INFO(0, open_mode)
3186:     ZEND_ARG_INFO(0, use_include_path)
3187:     ZEND_ARG_INFO(0, context)
3188:     ZEND_ARG_INFO(0, file_name)
3189:     ZEND_ARG_INFO(0, open_mode)
3190:     ZEND_ARG_INFO(0, use_include_path)
3191:     ZEND_ARG_INFO(0, context)
3192:     ZEND_ARG_INFO(0, file_name)
3193:     ZEND_ARG_INFO(0, open_mode)
3194:     ZEND_ARG_INFO(0, use_include_path)
3195:     ZEND_ARG_INFO(0, context)
3196:     ZEND_ARG_INFO(0, file_name)
3197:     ZEND_ARG_INFO(0, open_mode)
3198:     ZEND_ARG_INFO(0, use_include_path)
3199:     ZEND_ARG_INFO(0, context)
3200:     ZEND_ARG_INFO(0, file_name)
3201:     ZEND_ARG_INFO(0, open_mode)
3202:     ZEND_ARG_INFO(0, use_include_path)
3203:     ZEND_ARG_INFO(0, context)
3204:     ZEND_ARG_INFO(0, file_name)
3205:     ZEND_ARG_INFO(0, open_mode)
3206:     ZEND_ARG_INFO(0, use_include_path)
3207:     ZEND_ARG_INFO(0, context)
3208:     ZEND_ARG_INFO(0, file_name)
3209:     ZEND_ARG_INFO(0, open_mode)
3210:     ZEND_ARG_INFO(0, use_include_path)
3211:     ZEND_ARG_INFO(0, context)
3212:     ZEND_ARG_INFO(0, file_name)
3213:     ZEND_ARG_INFO(0, open_mode)
3214:     ZEND_ARG_INFO(0, use_include_path)
3215:     ZEND_ARG_INFO(0, context)
3216:     ZEND_ARG_INFO(0, file_name)
3217:     ZEND_ARG_INFO(0, open_mode)
3218:     ZEND_ARG_INFO(0, use_include_path)
3219:     ZEND_ARG_INFO(0, context)
3220:     ZEND_ARG_INFO(0, file_name)
3221:     ZEND_ARG_INFO(0, open_mode)
3222:     ZEND_ARG_INFO(0, use_include_path)
3223:     ZEND_ARG_INFO(0, context)
3224:     ZEND_ARG_INFO(0, file_name)
3225:     ZEND_ARG_INFO(0, open_mode)
3226:     ZEND_ARG_INFO(0, use_include_path)
3227:     ZEND_ARG_INFO(0, context)
3228:     ZEND_ARG_INFO(0, file_name)
3229:     ZEND_ARG_INFO(0, open_mode)
3230:     ZEND_ARG_INFO(0, use_include_path)
3231:     ZEND_ARG_INFO(0, context)
3232:     ZEND_ARG_INFO(0, file_name)
3233:     ZEND_ARG_INFO(0, open_mode)
3234:     ZEND_ARG_INFO(0, use_include_path)
3235:     ZEND_ARG_INFO(0, context)
3236:     ZEND_ARG_INFO(0, file_name)
3237:     ZEND_ARG_INFO(0, open_mode)
3238:     ZEND_ARG_INFO(0, use_include_path)
3239:     ZEND_ARG_INFO(0, context)
3240:     ZEND_ARG_INFO(0, file_name)
3241:     ZEND_ARG_INFO(0, open_mode)
3242:     ZEND_ARG_INFO(0, use_include_path)
3243:     ZEND_ARG_INFO(0, context)
3244:     ZEND_ARG_INFO(0, file_name)
3245:     ZEND_ARG_INFO(0, open_mode)
3246:     ZEND_ARG_INFO(0, use_include_path)
3247:     ZEND_ARG_INFO(0, context)
3248:     ZEND_ARG_INFO(0, file_name)
3249:     ZEND_ARG_INFO(0, open_mode)
3250:     ZEND_ARG_INFO(0, use_include_path)
3251:     ZEND_ARG_INFO(0, context)
3252:     ZEND_ARG_INFO(0, file_name)
3253:     ZEND_ARG_INFO(0, open_mode)
3254:     ZEND_ARG_INFO(0, use_include_path)
3255:     ZEND_ARG_INFO(0, context)
3256:     ZEND_ARG_INFO(0, file_name)
3257:     ZEND_ARG_INFO(0, open_mode)
3258:     ZEND_ARG_INFO(0, use_include_path)
3259:     ZEND_ARG_INFO(0, context)
3260:     ZEND_ARG_INFO(0, file_name)
3261:     ZEND_ARG_INFO(0, open_mode)
3262:     ZEND_ARG_INFO(0, use_include_path)
3263:     ZEND_ARG_INFO(0, context)
3264:     ZEND_ARG_INFO(0, file_name)
3265:     ZEND_ARG_INFO(0, open_mode)
3266:     ZEND_ARG_INFO(0, use_include_path)
3267:     ZEND_ARG_INFO(0, context)
3268:     ZEND_ARG_INFO(0, file_name)
3269:     ZEND_ARG_INFO(0, open_mode)
3270:     ZEND_ARG_INFO(0, use_include_path)
3271:     ZEND_ARG_INFO(0, context)
3272:     ZEND_ARG_INFO(0, file_name)
3273:     ZEND_ARG_INFO(0, open_mode)
3274:     ZEND_ARG_INFO(0, use_include_path)
3275:     ZEND_ARG_INFO(0, context)
3276:     ZEND_ARG_INFO(0, file_name)
3277:     ZEND_ARG_INFO(0, open_mode)
3278:     ZEND_ARG_INFO(0, use_include_path)
3279:     ZEND_ARG_INFO(0, context)
3280:     ZEND_ARG_INFO(0, file_name)
3281:     ZEND_ARG_INFO(0, open_mode)
3282:     ZEND_ARG_INFO(0, use_include_path)
3283:     ZEND_ARG_INFO(0, context)
3284:     ZEND_ARG_INFO(0, file_name)
3285:     ZEND_ARG_INFO(0, open_mode)
3286:     ZEND_ARG_INFO(0, use_include_path)
3287:     ZEND_ARG_INFO(0, context)
3288:     ZEND_ARG_INFO(0, file_name)
3289:     ZEND_ARG_INFO(0, open_mode)
3290:     ZEND_ARG_INFO(0, use_include_path)
3291:     ZEND_ARG_INFO(0, context)
3292:     ZEND_ARG_INFO(0, file_name)
3293:     ZEND_ARG_INFO(0, open_mode)
3294:     ZEND_ARG_INFO(0, use_include_path)
3295:     ZEND_ARG_INFO(0, context)
3296:     ZEND_ARG_INFO(0, file_name)
3297:     ZEND_ARG_INFO(0, open_mode)
3298:     ZEND_ARG_INFO(0, use_include_path)
3299:     ZEND_ARG_INFO(0, context)
3300:     ZEND_ARG_INFO(0, file_name)
3301:     ZEND_ARG_INFO(0, open_mode)
3302:     ZEND_ARG_INFO(0, use_include_path)
3303:     ZEND_ARG_INFO(0, context)
3304:     ZEND_ARG_INFO(0, file_name)
3305:     ZEND_ARG_INFO(0, open_mode)
3306:     ZEND_ARG_INFO(0, use_include_path)
3307:     ZEND_ARG_INFO(0, context)
3308:     ZEND_ARG_INFO(0, file_name)
3309:     ZEND_ARG_INFO(0, open_mode)
3310:     ZEND_ARG_INFO(0, use_include_path)
3311:     ZEND_ARG_INFO(0, context)
3312:     ZEND_ARG_INFO(0, file_name)
3313:     ZEND_ARG_INFO(0, open_mode)
3314:     ZEND_ARG_INFO(0, use_include_path)
3315:     ZEND_ARG_INFO(0, context)
3316:     ZEND_ARG_INFO(0, file_name)
3317:     ZEND_ARG_INFO(0, open_mode)
3318:     ZEND_ARG_INFO(0, use_include_path)
3319:     ZEND_ARG_INFO(0, context)
3320:     ZEND_ARG_INFO(0, file_name)
3321:     ZEND_ARG_INFO(0, open_mode)
3322:     ZEND_ARG_INFO(0, use_include_path)
3323:     ZEND_ARG_INFO(0, context)
3324:     ZEND_ARG_INFO(0, file_name)
3325:     ZEND_ARG_INFO(0, open_mode)
3326:     ZEND_ARG_INFO(0, use_include_path)
3327:     ZEND_ARG_INFO(0, context)
3328:     ZEND_ARG_INFO(0, file_name)
3329:     ZEND_ARG_INFO(0, open_mode)
3330:     ZEND_ARG_INFO(0, use_include_path)
3331:     ZEND_ARG_INFO(0, context)
3332:     ZEND_ARG_INFO(0, file_name)
3333:     ZEND_ARG_INFO(0, open_mode)
3334:     ZEND_ARG_INFO(0, use_include_path)
3335:     ZEND_ARG_INFO(0, context)
3336:     ZEND_ARG_INFO(0, file_name)
3337:     ZEND_ARG_INFO(0, open_mode)
3338:     ZEND_ARG_INFO(0, use_include_path)
3339:     ZEND_ARG_INFO(0, context)
3340:     ZEND_ARG_INFO(0, file_name)
3341:     ZEND_ARG_INFO(0, open_mode)
3342:     ZEND_ARG_INFO(0, use_include_path)
3343:     ZEND_ARG_INFO(0, context)
3344:     ZEND_ARG_INFO(0, file_name)
3345:     ZEND_ARG_INFO(0, open_mode)
3346:     ZEND_ARG_INFO(0, use_include_path)
3347:     ZEND_ARG_INFO(0, context)
3348:     ZEND_ARG_INFO(0, file_name)
3349:     ZEND_ARG_INFO(0, open_mode)
3350:     ZEND_ARG_INFO(0, use_include_path)
3351:     ZEND_ARG_INFO(0, context)
3352:     ZEND_ARG_INFO(0, file_name)
3353:     ZEND_ARG_INFO(0, open_mode)
3354:     ZEND_ARG_INFO(0, use_include_path)
3355:     ZEND_ARG_INFO(0, context)
3356:     ZEND_ARG_INFO(0, file_name)
3357:     ZEND_ARG_INFO(0, open_mode)
3358:     ZEND_ARG_INFO(0, use_include_path)
3359:     ZEND_ARG_INFO(0, context)
3360:     ZEND_ARG_INFO(0, file_name)
3361:     ZEND_ARG_INFO(0, open_mode)
3362:     ZEND_ARG_INFO(0, use_include_path)
3363:     ZEND_ARG_INFO(0, context)
3364:     ZEND_ARG_INFO(0, file_name)
3365:     ZEND_ARG_INFO(0, open_mode)
3366:     ZEND_ARG_INFO(0, use_include_path)
3367:     ZEND_ARG_INFO(0, context)
3368:     ZEND_ARG_INFO(0, file_name)
3369:     ZEND_ARG_INFO(0, open_mode)
3370:     ZEND_ARG_INFO(0, use_include_path)
3371:     ZEND_ARG_INFO(0, context)
3372:     ZEND_ARG_INFO(0, file_name)
3373:     ZEND_ARG_INFO(0, open_mode)
3374:     ZEND_ARG_INFO(0, use_include_path)
3375:     ZEND_ARG_INFO(0, context)
3376:     ZEND_ARG_INFO(0, file_name)
3377:     ZEND_ARG_INFO(0, open_mode)
3378:     ZEND_ARG_INFO(0, use_include_path)
3379:     ZEND_ARG_INFO(0, context)
3380:     ZEND_ARG_INFO(0, file_name)
3381:     ZEND_ARG_INFO(0, open_mode)
3382:     ZEND_ARG_INFO(0, use_include_path)
3383:     ZEND_ARG_INFO(0, context)
3384:     ZEND_ARG_INFO(0, file_name)
3385:     ZEND_ARG_INFO(0, open_mode)
3386:     ZEND_ARG_INFO(0, use_include_path)
3387:     ZEND_ARG_INFO(0, context)
3388:     ZEND_ARG_INFO(0, file_name)
3389:     ZEND_ARG_INFO(0, open_mode)
3390:     ZEND_ARG_INFO(0, use_include_path)
3391:     ZEND_ARG_INFO(0, context)
3392:     ZEND_ARG_INFO(0, file_name)
3393:     ZEND_ARG_INFO(0, open_mode)
3394:     ZEND_ARG_INFO(0, use_include_path)
3395:     ZEND_ARG_INFO(0, context)
3396:     ZEND_ARG_INFO(0, file_name)
3397:     ZEND_ARG_INFO(0, open_mode)
3398:     ZEND_ARG_INFO(0, use_include_path)
3399:     ZEND_ARG_INFO(0, context)
3400:     ZEND_ARG_INFO(0, file_name)
3401:     ZEND_ARG_INFO(0, open_mode)
3402:     ZEND_ARG_INFO(0, use_include_path)
3403:     ZEND_ARG_INFO(0, context)
3404:     ZEND_ARG_INFO(0, file_name)
3405:     ZEND_ARG_INFO(0, open_mode)
3406:     ZEND_ARG_INFO(0, use_include_path)
3407:     ZEND_ARG_INFO(0, context)
3408:     ZEND_ARG_INFO(0, file_name)
3409:     ZEND_ARG_INFO(0, open_mode)
3410:     ZEND_ARG_INFO(0, use_include_path)
3411:     ZEND_ARG_INFO(0, context)
3412:     ZEND_ARG_INFO(0, file_name)
3413:     ZEND_ARG_INFO(0, open_mode)
3414:     ZEND_ARG_INFO(0, use_include_path)
3415:     ZEND_ARG_INFO(0, context)
3416:     ZEND_ARG_INFO(0, file_name)
3417:     ZEND_ARG_INFO(0, open_mode)
3418:     ZEND_ARG_INFO(0, use_include_path)
3419:     ZEND_ARG_INFO(0, context)
3420:     ZEND_ARG_INFO(0, file_name)
3421:     ZEND_ARG_INFO(0, open_mode)
3422:     ZEND_ARG_INFO(0, use_include_path)
3423:     ZEND_ARG_INFO(0, context)
3424:     ZEND_ARG_INFO(0, file_name)
3425:     ZEND_ARG_INFO(0, open_mode)
3426:     ZEND_ARG_INFO(0, use_include_path)
3427:     ZEND_ARG_INFO(0, context)
3428:     ZEND_ARG_INFO(0, file_name)
3429:     ZEND_ARG_INFO(0, open_mode)
3430:     ZEND_ARG_INFO(0, use_include_path)
3431:     ZEND_ARG_INFO(0, context)
3432:     ZEND_ARG_INFO(0, file_name)
3433:     ZEND_ARG_INFO(0, open_mode)
3434:     ZEND_ARG_INFO(0, use_include_path)
3435:     ZEND_ARG_INFO(0, context)
3436:     ZEND_ARG_INFO(0, file_name)
3437:     ZEND_ARG_INFO(0, open_mode)
3438:     ZEND_ARG_INFO(0, use_include_path)
3439:     ZEND_ARG_INFO(0, context)
3440:     ZEND_ARG_INFO(0, file_name)
3441:     ZEND_ARG_INFO(0, open_mode)
3442:     ZEND_ARG_INFO(0, use_include_path)
3443:     ZEND_ARG_INFO(0, context)
3444:     ZEND_ARG_INFO(0, file_name)
3445:     ZEND_ARG_INFO(0, open_mode)
3446:     ZEND_ARG_INFO(0, use_include_path)
3447:     ZEND_ARG_INFO(0, context)
3448:     ZEND_ARG_INFO(0, file_name)
3449:     ZEND_ARG_INFO(0, open_mode)
3450:     ZEND_ARG_INFO(0, use_include_path)
3451
```



```
3005: ZEND_ARG_INFO(0, escape)
3006: ZEND_END_ARG_INFO()
3007:
3008: ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_fputcsv, 0, 0, 1)
3009: ZEND_ARG_INFO(0, fields)
3010: ZEND_ARG_INFO(0, delimiter)
3011: ZEND_ARG_INFO(0, enclosure)
3012: ZEND_ARG_INFO(0, escape)
3013: ZEND_END_ARG_INFO()
3014:
3015: ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_flock, 0, 0, 1)
3016: ZEND_ARG_INFO(0, operation)
3017: ZEND_ARG_INFO(1, wouldblock)
3018: ZEND_END_ARG_INFO()
3019:
3020: ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_fseek, 0, 0, 1)
3021: ZEND_ARG_INFO(0, pos)
3022: ZEND_ARG_INFO(0, whence)
3023: ZEND_END_ARG_INFO()
3024:
3025: ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_fgetas, 0, 0, 0)
3026: ZEND_ARG_INFO(0, allowable_tags)
3027: ZEND_END_ARG_INFO()
3028:
3029: ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_fscanf, 0, 0, 1)
3030: ZEND_ARG_INFO(0, format)
3031: ZEND_ARG_VARIADIC_INFO(1, vars)
3032: ZEND_END_ARG_INFO()
3033:
3034: ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_fwrite, 0, 0, 1)
3035: ZEND_ARG_INFO(0, str)
3036: ZEND_ARG_INFO(0, length)
3037: ZEND_END_ARG_INFO()
3038:
3039: ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_fread, 0, 0, 1)
3040: ZEND_ARG_INFO(0, length)
3041: ZEND_END_ARG_INFO()
3042:
3043: ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_ftruncate, 0, 0, 1)
3044: ZEND_ARG_INFO(0, size)
3045: ZEND_END_ARG_INFO()
3046:
3047: ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_seek, 0, 0, 1)
3048: ZEND_ARG_INFO(0, line_pos)
3049: ZEND_END_ARG_INFO()
3050:
3051: static const zend_function_entry spl_SplFileInfo_functions[] = {
3052:     SPL_ME(SplFileInfo, __construct, arginfo_file_object__construct, ZEND_ACC_PUBLIC)
3053:     SPL_ME(SplFileInfo, rewind, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3054:     SPL_ME(SplFileInfo, eof, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3055:     SPL_ME(SplFileInfo, valid, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3056:     SPL_ME(SplFileInfo, fgetc, arginfo_file_object_fgetc, ZEND_ACC_PUBLIC)
3057:     SPL_ME(SplFileInfo, fgetcsv, arginfo_file_object_fgetcsv, ZEND_ACC_PUBLIC)
3058:     SPL_ME(SplFileInfo, fputc, arginfo_file_object_fputc, ZEND_ACC_PUBLIC)
3059:     SPL_ME(SplFileInfo, setcsvControl, arginfo_file_object_fputcsv, ZEND_ACC_PUBLIC)
3060:     SPL_ME(SplFileInfo, getcsvControl, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3061:     SPL_ME(SplFileInfo, flock, arginfo_file_object_flock, ZEND_ACC_PUBLIC)
3062:     SPL_ME(SplFileInfo, fflush, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3063:     SPL_ME(SplFileInfo, ftruncate, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3064:     SPL_ME(SplFileInfo, fseek, arginfo_file_object_fseek, ZEND_ACC_PUBLIC)
3065:     SPL_ME(SplFileInfo, fgetc, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3066:     SPL_ME(SplFileInfo, fpassthru, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3067:     SPL_ME(SplFileInfo, fgetas, arginfo_file_object_fgetas, ZEND_ACC_PUBLIC)
3068:     SPL_ME(SplFileInfo, fscanf, arginfo_file_object_fscanf, ZEND_ACC_PUBLIC)
3069:     SPL_ME(SplFileInfo, fwrite, arginfo_file_object_fwrite, ZEND_ACC_PUBLIC)
3070:     SPL_ME(SplFileInfo, fread, arginfo_file_object_fread, ZEND_ACC_PUBLIC)
3071:     SPL_ME(SplFileInfo, fstat, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3072:     SPL_ME(SplFileInfo, truncate, arginfo_file_object_ftruncate, ZEND_ACC_PUBLIC)
3073:     SPL_ME(SplFileInfo, current, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3074:     SPL_ME(SplFileInfo, key, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3075:     SPL_ME(SplFileInfo, next, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3076:     SPL_ME(SplFileInfo, setFlags, arginfo_file_object_setFlags, ZEND_ACC_PUBLIC)
3077:     SPL_ME(SplFileInfo, getFlags, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3078:     SPL_ME(SplFileInfo, setMaxLineLen, arginfo_file_object_setMaxLineLen, ZEND_ACC_PUBLIC)
3079:     SPL_ME(SplFileInfo, getMaxLineLen, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3080:     SPL_ME(SplFileInfo, getChilden, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3081:     SPL_ME(SplFileInfo, getChilden, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3082:     SPL_ME(SplFileInfo, seek, arginfo_file_object_seek, ZEND_ACC_PUBLIC)
3083:     /* mappings */
3084:     SPL_ME(SplFileInfo, getCurrentLine, SplFileInfo, fgetc, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3085:     SPL_ME(SplFileInfo, __toString, SplFileInfo, current, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3086:     PHP_FE_END
3087: };
3088:
3089: ZEND_BEGIN_ARG_INFO_EX(arginfo_temp_file_object__construct, 0, 0, 0)
3090: ZEND_ARG_INFO(0, max_memory)
3091: ZEND_END_ARG_INFO()
3092:
3093: static const zend_function_entry spl_SplTempFileObject_functions[] = {
3094:     SPL_ME(SplTempFileObject, __construct, arginfo_temp_file_object__construct, ZEND_ACC_PUBLIC)
3095:     PHP_FE_END
3096: };
3097: /* }}} */
3098:
3099: /* {{{ PHP_MINIT_FUNCTION(spl_directory)
3100: */
3101: /* {{{ PHP_MINIT_FUNCTION(spl_directory)
3102: */
3103: REGISTER_SPL_STD_CLASS_EX(SplFileInfo, spl_filesystem_object_new, spl_SplFileInfo_functions);
3104: memory(spl_filesystem_object_handlers, zend_get_object_handlers(), sizeof(zend_object_handlers));
3105: spl_filesystem_object_handlers.offset = XOffsetOf(spl_filesystem_object, std);
3106: spl_filesystem_object_handlers.clone_obj = spl_filesystem_object_clone;
3107: spl_filesystem_object_handlers.cast_object = spl_filesystem_object_cast;
3108: spl_filesystem_object_handlers.get_debug_info = spl_filesystem_object_get_debug_info;
3109: spl_filesystem_object_handlers.dtor_obj = spl_filesystem_object_destroy_object;
3110: spl_filesystem_object_handlers.free_obj = spl_filesystem_object_free_storage;
3111: spl_ow_SplFileInfo->serialize = zend_class_serialize_deny;
3112: spl_ow_SplFileInfo->unserialize = zend_class_unserialize_deny;
3113:
3114:
3115: REGISTER_SPL_SUB_CLASS_EX(DirectoryIterator, SplFileInfo, spl_filesystem_object_new, spl_DirectoryIterator_functions);
3116: zend_class_implements(spl_ow_DirectoryIterator, 1, zend_ow_iterator);
3117: REGISTER_SPL_IMPLMENTS(DirectoryIterator, SeekableIterator);
3118:
3119: spl_ow_DirectoryIterator->get_iterator = spl_filesystem_dir_get_iterator;
3120:
3121: REGISTER_SPL_SUB_CLASS_EX(FilesystemIterator, DirectoryIterator, spl_filesystem_object_new, spl_filesystem_iterator_functions);
3122:
3123: REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "CURRENT_MODE_MASK", SPL_FILE_DIR_CURRENT_MODE_MASK);
3124: REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "CURRENT_AS_PATHNAME", SPL_FILE_DIR_CURRENT_AS_PATHNAME);
3125: REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "CURRENT_AS_FILEINFO", SPL_FILE_DIR_CURRENT_AS_FILEINFO);
3126: REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "CURRENT_AS_SELF", SPL_FILE_DIR_CURRENT_AS_SELF);
3127: REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "KEY_MODE_MASK", SPL_FILE_DIR_KEY_MODE_MASK);
3128: REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "KEY_AS_PATHNAME", SPL_FILE_DIR_KEY_AS_PATHNAME);
3129: REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "FOLLOW_SYMLINKS", SPL_FILE_DIR_FOLLOW_SYMLINKS);
3130: REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "KEY_AS_FILENAME", SPL_FILE_DIR_KEY_AS_FILENAME);
3131: REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "NEW_CURRENT_AS_KEY", SPL_FILE_DIR_KEY_AS_FILENAME(SPL_FILE_DIR_CURRENT_AS_FILEINFO);
3132: REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "OTHER_MODE_MASK", SPL_FILE_DIR_OTHERS_MASK);
3133: REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "SKIP_DOTS", SPL_FILE_DIR_SKIPDOTS);
3134: REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "UNIX_PATHS", SPL_FILE_DIR_UNIXPATHS);
3135:
3136: spl_ow_FilesystemIterator->get_iterator = spl_filesystem_tree_get_iterator;
3137:
3138: REGISTER_SPL_SUB_CLASS_EX(RecursiveDirectoryIterator, FilesystemIterator, spl_filesystem_object_new, spl_RecursiveDirectoryIterator_functions);
3139: REGISTER_SPL_IMPLMENTS(RecursiveDirectoryIterator, RecursiveIterator);
3140:
3141: memory(spl_filesystem_object_check_handlers, spl_filesystem_object_handlers, sizeof(zend_object_handlers));
3142: spl_filesystem_object_check_handlers.get_method = spl_filesystem_object_get_method_check;
3143:
3144: #ifdef HAVE_GLOB
3145: REGISTER_SPL_SUB_CLASS_EX(GlobIterator, FilesystemIterator, spl_filesystem_object_new_check, spl_GlobIterator_functions);
3146: REGISTER_SPL_IMPLMENTS(GlobIterator, Countable);
3147: #endif
3148:
3149: REGISTER_SPL_SUB_CLASS_EX(SplFileInfo, SplFileInfo, spl_filesystem_object_new_check, spl_SplFileInfo_functions);
3150: REGISTER_SPL_IMPLMENTS(SplFileInfo, RecursiveIterator);
3151: REGISTER_SPL_IMPLMENTS(SplFileInfo, SeekableIterator);
3152:
3153: REGISTER_SPL_CLASS_CONST_LONG(SplFileInfo, "DROP_NEW_LINE", SPL_FILE_OBJECT_DROP_NEW_LINE);
3154: REGISTER_SPL_CLASS_CONST_LONG(SplFileInfo, "READ_AHEAD", SPL_FILE_OBJECT_READ_AHEAD);
3155: REGISTER_SPL_CLASS_CONST_LONG(SplFileInfo, "READ_BUFFER", SPL_FILE_OBJECT_READ_BUFFER);
3156: REGISTER_SPL_CLASS_CONST_LONG(SplFileInfo, "READ_CSV", SPL_FILE_OBJECT_READ_CSV);
3157:
3158: REGISTER_SPL_SUB_CLASS_EX(SplTempFileObject, SplFileInfo, spl_filesystem_object_new_check, spl_SplTempFileObject_functions);
3159: #define SUCCESS;
3160: }
3161: /* }}} */
3162:
3163: /*
3164:  * Local variables:
3165:  * tab-width: 4
3166:  * c-basic-offset: 4
3167:  * End:
3168:  * vim600: not set4 ts=4 fdm=marker
3169:  * vim600: not set4 ts=4
3170: */
```

```
1: /*
2:  * -----
3:  * | PHP Version 7 |
4:  * -----
5:  * | Copyright (c) 1997-2018 The PHP Group |
6:  * -----
7:  * | This source file is subject to version 3.01 of the PHP license, |
8:  * | that is bundled with this package in the file LICENSE, and is |
9:  * | available through the world-wide-web at the following url: |
10:  * | https://www.php.net/license/3.01.txt |
11:  * | If you did not receive a copy of the PHP license and are unable to |
12:  * | obtain it through the world-wide-web, please send a note to |
13:  * | license@php.net so we can mail you a copy immediately. |
14:  * -----
15:  * | Authors: Marcus Boerger <helly@php.net> |
16:  * -----
17:  */
18:
19: /* $Id$ */
20:
21: #ifndef SPL_ITERATORS_H
22: #define SPL_ITERATORS_H
23:
24: #include "php.h"
25: #include "php_spl.h"
26: #if HAVE_BUNDLED_PCRE
27: #include "ext/pcre/php_pcre.h"
28: #endif
29:
30: #define spl_ow_traversable send_ow_traversable
31: #define spl_ow_iterator send_ow_iterator
32: #define spl_ow_aggregate send_ow_aggregate
33: #define spl_ow_arrayaccess send_ow_arrayaccess
34: #define spl_ow_serializable send_ow_serializable
35: #define spl_ow_countable send_ow_countable
36:
37: #define PHPAPI zend_class_entry *spl_ow_recursiveiterator;
38: #define PHPAPI zend_class_entry *spl_ow_recursiveiteratoriterator;
39: #define PHPAPI zend_class_entry *spl_ow_recursiveiteratoriterator;
40: #define PHPAPI zend_class_entry *spl_ow_filteriterator;
41: #define PHPAPI zend_class_entry *spl_ow_recursivefilteriterator;
42: #define PHPAPI zend_class_entry *spl_ow_parentiterator;
43: #define PHPAPI zend_class_entry *spl_ow_weakableiterator;
44: #define PHPAPI zend_class_entry *spl_ow_limititerator;
45: #define PHPAPI zend_class_entry *spl_ow_cachingiterator;
46: #define PHPAPI zend_class_entry *spl_ow_recursivecachingiterator;
47: #define PHPAPI zend_class_entry *spl_ow_outertiterator;
48: #define PHPAPI zend_class_entry *spl_ow_iteratoriterator;
49: #define PHPAPI zend_class_entry *spl_ow_mohanditerator;
50: #define PHPAPI zend_class_entry *spl_ow_infiniteiterator;
51: #define PHPAPI zend_class_entry *spl_ow_emptyiterator;
52: #define PHPAPI zend_class_entry *spl_ow_appenditerator;
53: #define PHPAPI zend_class_entry *spl_ow_requeiterator;
54: #define PHPAPI zend_class_entry *spl_ow_recursiverequeiterator;
55: #define PHPAPI zend_class_entry *spl_ow_callbackfilteriterator;
56: #define PHPAPI zend_class_entry *spl_ow_recursivecallbackfilteriterator;
57:
58: #define PHP_MINIT_FUNCTION(spl_iterators);
59:
60: #define PHP_FUNCTION(iterator_to_array);
61: #define PHP_FUNCTION(iterator_count);
62: #define PHP_FUNCTION(iterator_apply);
63:
64: typedef enum {
65:     DIT_Default = 0,
66:     DIT_Filteriterator = DIT_Default,
67:     DIT_RecursiveFilteriterator = DIT_Default,
68:     DIT_Parentiterator = DIT_Default,
69:     DIT_Limititerator,
70:     DIT_Cachingiterator,
71:     DIT_RecursiveCachingiterator,
72:     DIT_Iteratoriterator,
73:     DIT_Mohanditerator,
74:     DIT_Infiniteiterator,
75:     DIT_Appenditerator,
76: #if HAVE_PCRE
77:     DIT_Requeiterator,
78:     DIT_RecursiveRequeiterator,
79: #endif
80:     DIT_CallbackFilteriterator,
81:     DIT_RecursiveCallbackFilteriterator,
82:     DIT_Unknown = 0
83: } dual_it_type;
84:
85: typedef enum {
86:     RIT_Default = 0,
87:     RIT_RecursiveIteratorIterator = RIT_Default,
88:     RIT_RecursiveTreeIterator,
89:     RIT_Unknown = 0
90: } recursive_it_type;
91:
92: enum {
93:     /* public */
94:     CIT_CALL_POSTING = 0x00000001,
95:     CIT_YOSTRING_USER_KEY = 0x00000002,
96:     CIT_YOSTRING_USER_CURRENT = 0x00000004,
97:     CIT_YOSTRING_USER_INNER = 0x00000008,
98:     CIT_CATCH_GET_CHILD = 0x00000010,
99:     CIT_FULL_CACHE = 0x00000100,
100:     CIT_PUBLIC = 0x000000FFFF,
101:     /* private */
102:     CIT_VALID = 0x00010000,
103:     CIT_HAS_CHILDREN = 0x00020000
104: };
105:
106: enum {
107:     /* public */
108:     REGIT_USER_KEY = 0x00000001,
109:     REGIT_INVERTED = 0x00000002
110: };
111:
112: typedef enum {
113:     REGIT_MODE_MATCH,
114:     REGIT_MODE_GET_MATCH,
115:     REGIT_MODE_ALL_MATCHES,
116:     REGIT_MODE_SPLIT,
117:     REGIT_MODE_REPLACE,
118:     REGIT_MODE_MAX
119: } regex_mode;
120:
121: typedef struct _spl_chfilter_it_intern {
122:     zend_fcall_info fci;
123:     zend_fcall_info_cache fcci;
124:     zend_object *obj;
125: } _spl_chfilter_it_intern;
126:
127: typedef struct _spl_dual_it_object {
128:     struct {
129:         zval zobj;
130:         zend_class_entry *ce;
131:         zend_object *obj;
132:         zend_object_iterator *iterator;
133:     } inner;
134:     struct {
135:         zval data;
136:         zval key;
137:         zend_long pos;
138:     } current;
139:     dual_it_type dit_type;
140:     union {
141:         struct {
142:             zend_long offset;
143:             zend_long count;
144:         } limit;
145:         struct {
146:             zend_long flags; /* CIT_* */
147:             zval src;
148:             zval children;
149:             zval scache;
150:         } caching;
151:         struct {
152:             zval array;
153:             zend_object_iterator *iterator;
154:         } append;
155: #if HAVE_PCRE
156:         struct {
157:             zend_long flags;
158:             zend_long preg_flags;
159:             pcre_cache_entry *pcre;
160:             zend_string *regex;
161:             regex_mode mode;
162:             int use_flags;
163:         } regex;
164: #endif
165:     } _spl_chfilter_it_intern *chfilter;
166: } _spl_dual_it_object;
167:
168: static inline spl_dual_it_object *spl_dual_it_from_obj(zend_object *obj) {
169:     return (spl_dual_it_object *) ((char *) (obj) - XOFFSETOF(spl_dual_it_object, std));
170: }
171:
172: #define SPL_DUAL_IT_P(xv) spl_dual_it_from_obj((zend_object *) (xv))
173:
174: #define SPL_DUAL_IT_P(xv) spl_dual_it_from_obj((zend_object *) (xv))
175:
176: typedef int (*spl_iterator_apply_func_t)(zend_object_iterator *iter, void *puser);
177:
178: PHPAPI int spl_iterator_apply(zval *obj, spl_iterator_apply_func_t apply_func, void *puser);
179:
180: #endif /* SPL_ITERATORS_H */
181:
182: /*
183:  * Local Variables:
184:  * * tab-width: 4
185:  * * tab-width: 4
186:  * * End:
187:  * * vim600: fdm=marker
188:  * * vim: noet sw=4 ts=4
189:  */
```

189: /\*



[illegible]

```

377:         case BP_VAR_RSW:
378:             zend_error(E_NOTICE, "Undefined offset: " ZEND_LONG_FMT, index);
379:         case BP_VAR_W: {
380:             zval *value;
381:             ZVAL_UNDEF(value);
382:             retval = zend_hash_index_update(ht, index, value);
383:         }
384:     }
385: }
386: return retval;
387: case IS_REFERENCE:
388:     ZVAL_DEREF(offset);
389:     goto try_again;
390: default:
391:     zend_error(E_WARNING, "Illegal offset type");
392:     return (type == BP_VAR_P || type == BP_VAR_RSW ?
393:         &EG(uninitialized_val) : &EG(uninitialized_val));
394: }
395: /* }}} */
396:
397: static int spl_array_has_dimension(zval *object, zval *offset, int check_empty);
398:
399: static zval *spl_array_read_dimension_ex(int check_inherited, zval *object, zval *offset, int type, zval *rv) /* {{{ */
400: {
401:     spl_array_object *intern = Z_SPLARRAY_P(object);
402:     zval *ret;
403:
404:     if (check_inherited &&
405:         (intern->fptr_offset_get || (type == BP_VAR_IS && intern->fptr_offset_has))) {
406:         if (type == BP_VAR_IS) {
407:             if (!spl_array_has_dimension(object, offset, 0)) {
408:                 return &EG(uninitialized_val);
409:             }
410:         }
411:
412:         if (intern->fptr_offset_get) {
413:             zval tmp;
414:             if (!offset) {
415:                 ZVAL_UNDEF(&tmp);
416:                 offset = &tmp;
417:             } else {
418:                 SEPARATE_ARG_1_PTR(offset);
419:             }
420:
421:             zend_call_method_with_1_params(object, Z_OBJCE_P(object), &intern->fptr_offset_get, "offsetGet", rv, offset);
422:             zval_ptr_dtor(offset);
423:
424:             if (!IS_UNDEF_P(rv)) {
425:                 return rv;
426:             }
427:             return &EG(uninitialized_val);
428:         }
429:
430:         ret = spl_array_get_dimension_ptr(check_inherited, intern, offset, type);
431:
432:         /* When in a write context,
433:          * * ZE has to be fooled into thinking this is in a reference set
434:          * * by separating (if necessary) and returning as IS_REFERENCE (with refcount == 1)
435:          */
436:
437:         if (type == BP_VAR_W || type == BP_VAR_RSW || type == BP_VAR_UNSET) &&
438:             IS_UNDEF_P(ret) &&
439:             EXPECTED(ret != &EG(uninitialized_val)) {
440:             ZVAL_UNDEF(ret, ret);
441:         }
442:
443:         return ret;
444:     /* }}} */
445:
446:     static zval *spl_array_read_dimension(zval *object, zval *offset, int type, zval *rv) /* {{{ */
447:     {
448:         return spl_array_read_dimension_ex(1, object, offset, type, rv);
449:     /* }}} */
450:
451:     static void spl_array_write_dimension_ex(int check_inherited, zval *object, zval *offset, zval *value) /* {{{ */
452:     {
453:         spl_array_object *intern = Z_SPLARRAY_P(object);
454:         zend_long index;
455:         HashTable *ht;
456:
457:         if (check_inherited && intern->fptr_offset_set) {
458:             zval tmp;
459:
460:             if (!offset) {
461:                 ZVAL_NULL(&tmp);
462:                 offset = &tmp;
463:             } else {
464:                 SEPARATE_ARG_1_PTR(offset);
465:             }
466:             zend_call_method_with_2_params(object, Z_OBJCE_P(object), &intern->fptr_offset_set, "offsetSet", NULL, offset, value);
467:             zval_ptr_dtor(offset);
468:             return;
469:         }
470:
471:         if (intern->nApplyCount > 0) {
472:             zend_error(E_WARNING, "Modification of ArrayObject during sorting is prohibited");
473:             return;
474:         }
475:
476:         Z_TRY_ADDREF_P(value);
477:         if (!offset) {
478:             ht = spl_array_get_hash_table(intern);
479:             zend_hash_next_index_insert(ht, value);
480:             return;
481:         }
482:
483:         try_again:
484:         switch (Z_TYPE_P(offset)) {
485:             case IS_STRING:
486:                 ht = spl_array_get_hash_table(intern);
487:                 zend_symtable_update_and(ht, Z_STR_P(offset), value);
488:                 return;
489:             case IS_DOUBLE:
490:                 index = (zend_long)Z_DVAL_P(offset);
491:                 goto num_index;
492:             case IS_RESOURCE:
493:                 index = Z_RES_HANDLE_P(offset);
494:                 goto num_index;
495:             case IS_FALSE:
496:                 index = 0;
497:                 goto num_index;
498:             case IS_TRUE:
499:                 index = 1;
500:                 goto num_index;
501:             case IS_LONG:
502:                 index = Z_LVAL_P(offset);
503:                 goto num_index;
504:             case IS_NULL:
505:                 ht = spl_array_get_hash_table(intern);
506:                 zend_hash_index_update(ht, index, value);
507:                 return;
508:             case IS_REFERENCE:
509:                 ht = spl_array_get_hash_table(intern);
510:                 zend_hash_next_index_insert(ht, value);
511:                 return;
512:             case IS_STRING:
513:                 ZVAL_DEREF(offset);
514:                 goto try_again;
515:             default:
516:                 zend_error(E_WARNING, "Illegal offset type");
517:                 zval_ptr_dtor(value);
518:                 return;
519:         /* }}} */
520:
521:     static void spl_array_write_dimension(zval *object, zval *offset, zval *value) /* {{{ */
522:     {
523:         spl_array_write_dimension_ex(1, object, offset, value);
524:     /* }}} */
525:
526:     static void spl_array_unset_dimension_ex(int check_inherited, zval *object, zval *offset) /* {{{ */
527:     {
528:         zend_long index;
529:         HashTable *ht;
530:         spl_array_object *intern = Z_SPLARRAY_P(object);
531:
532:         if (check_inherited && intern->fptr_offset_del) {
533:             SEPARATE_ARG_1_PTR(offset);
534:             zend_call_method_with_1_params(object, Z_OBJCE_P(object), &intern->fptr_offset_del, "offsetUnset", NULL, offset);
535:             zval_ptr_dtor(offset);
536:             return;
537:         }
538:
539:         if (intern->nApplyCount > 0) {
540:             zend_error(E_WARNING, "Modification of ArrayObject during sorting is prohibited");
541:             return;
542:         }
543:
544:         try_again:
545:         switch (Z_TYPE_P(offset)) {
546:             case IS_STRING:
547:                 ht = spl_array_get_hash_table(intern);
548:                 if (ht == &EG(symtbl_table)) {
549:                     if (zend_delete_global_variable(Z_STR_P(offset))) {
550:                         zend_error(E_NOTICE, "Undefined index: %s", Z_STRVAL_P(offset));
551:                     }
552:                 } else {
553:                     zval *data = zend_symtable_find(ht, Z_STR_P(offset));
554:
555:                     if (data) {
556:                         if (Z_TYPE_P(data) == IS_INDIRECT) {
557:                             data = Z_INDIRECT_P(data);
558:                             if (IS_UNDEF_P(data) == IS_UNDEF) {
559:                                 zend_error(E_NOTICE, "Undefined index: %s", Z_STRVAL_P(offset));
560:                             } else {
561:                                 zval_ptr_dtor(data);
562:                             }
563:                         }
564:                         ZVAL_UNDEF(data);
565:                         HT_FLAGS(ht) |= HASH_FLAG_HAS_EMPTY_VAL;
566:                         zend_hash_move_forward_ex(ht, spl_array_get_pos_ptr(ht, intern));
567:                     }
568:                 }
569:             }
570:         }
571:
572:         if (spl_array_is_array_protected(intern, ht)) {
573:             }
574:         } else if (zend_symtable_del(ht, Z_STR_P(offset)) == FAILURE) {
575:             zend_error(E_NOTICE, "Undefined index: %s", Z_STRVAL_P(offset));
576:         }
577:     }
578:     } else {
579:         zend_error(E_NOTICE, "Undefined index: %s", Z_STRVAL_P(offset));
580:     }
581:     }
582:     }
583:     }
584:     }
585:     }
586:     }
587:     }
588:     }
589:     }
590:     }
591:     }
592:     }
593:     }
594:     }
595:     }
596:     }
597:     }
598:     }
599:     }
600:     }
601:     }
602:     }
603:     }
604:     }
605:     }
606:     }
607:     }
608:     }
609:     }
610:     }
611:     }
612:     }
613:     }
614:     }
615:     }
616:     }
617:     }
618:     }
619:     }
620:     }
621:     }
622:     }
623:     }
624:     }
625:     }
626:     }
627:     }
628:     }
629:     }
630:     }
631:     }
632:     }
633:     }
634:     }
635:     }
636:     }
637:     }
638:     }
639:     }
640:     }
641:     }
642:     }
643:     }
644:     }
645:     }
646:     }
647:     }
648:     }
649:     }
650:     }
651:     }
652:     }
653:     }
654:     }
655:     }
656:     }
657:     }
658:     }
659:     }
660:     }
661:     }
662:     }
663:     }
664:     }
665:     }
666:     }
667:     }
668:     }
669:     }
670:     }
671:     }
672:     }
673:     }
674:     }
675:     }
676:     }
677:     }
678:     }
679:     }
680:     }
681:     }
682:     }
683:     }
684:     }
685:     }
686:     }
687:     }
688:     }
689:     }
690:     }
691:     }
692:     }
693:     }
694:     }
695:     }
696:     }
697:     }
698:     }
699:     }
700:     }
701:     }
702:     }
703:     }
704:     }
705:     }
706:     }
707:     }
708:     }
709:     }
710:     }
711:     }
712:     }
713:     }
714:     }
715:     }
716:     }
717:     }
718:     }
719:     }
720:     }
721:     }
722:     }
723:     }
724:     }
725:     }
726:     }
727:     }
728:     }
729:     }
730:     }
731:     }
732:     }
733:     }
734:     }
735:     }
736:     }
737:     }
738:     }
739:     }
740:     }
741:     }
742:     }
743:     }
744:     }
745:     }
746:     }
747:     }
748:     }
749:     }
750:     }
751:     }
752:     }

```

```

753: if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "ss", &index, &value) == FAILURE) {
754:     return;
755: }
756: spl_array_write_dimension_ex(0, getThis(), index, value);
757: } /* }}} */
758:
759: void spl_array_iterator_append(zval *object, zval *append_value) /* {{{ */
760: {
761:     spl_array_object *intern = Z_SPLARRAY_P(object);
762:     HashTable *aht = spl_array_get_hash_table(intern);
763:
764:     if (!aht) {
765:         php_error_docref(NULL, E_NOTICE, "Array was modified outside object and is no longer an array");
766:         return;
767:     }
768:
769:     if (spl_array_is_object(intern) {
770:         zend_throw_error(NULL, "Cannot append properties to objects, use %s::offsetSet() instead", ZSTR_VAL(Z_OBJCE_P(object)->name));
771:         return;
772:     }
773:
774:     spl_array_write_dimension(object, NULL, append_value);
775: } /* }}} */
776:
777: /* {{{ proto void ArrayObject::append(mixed $value)
778:    proto void ArrayIterator::append(mixed $value)
779:    Appends the value (cannot be called for objects). */
780: #define SPL_METHOD(Array, append)
781:
782: zval *value;
783:
784: if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "s", &value) == FAILURE) {
785:     return;
786: }
787: spl_array_iterator_append(getThis(), value);
788: } /* }}} */
789:
790: /* {{{ proto void ArrayObject::offsetUnset(mixed $index)
791:    proto void ArrayIterator::offsetUnset(mixed $index)
792:    Unsets the value at the specified $index. */
793: #define SPL_METHOD(Array, offsetUnset)
794:
795: zval *index;
796: if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "s", &index) == FAILURE) {
797:     return;
798: }
799: spl_array_unset_dimension_ex(0, getThis(), index);
800: } /* }}} */
801:
802: /* {{{ proto array ArrayObject::getArrayCopy()
803:    proto array ArrayIterator::getArrayCopy()
804:    Returns a copy of the contained array */
805: #define SPL_METHOD(Array, getArrayCopy)
806:
807: zval *object = getThis();
808: spl_array_object *intern = Z_SPLARRAY_P(object);
809:
810: RETURN_ARRAY(zend_array_dup(spl_array_get_hash_table(intern)));
811: } /* }}} */
812:
813: static HashTable *spl_array_get_properties(zval *object) /* {{{ */
814: {
815:     spl_array_object *intern = Z_SPLARRAY_P(object);
816:
817:     if ((intern->var_flags & SPL_ARRAY_STD_PROP_LIST) {
818:         if ((intern->std_properties) {
819:             rebuild_object_properties(&intern->std);
820:         }
821:         return intern->std_properties;
822:     }
823:
824:     return spl_array_get_hash_table(intern);
825: } /* }}} */
826:
827: static HashTable * spl_array_get_debug_info(zval *obj, int *is_temp) /* {{{ */
828: {
829:     zval *storage;
830:     zend_string *name;
831:     zend_string *key;
832:     spl_array_object *intern = Z_SPLARRAY_P(obj);
833:
834:     if ((intern->std_properties) {
835:         rebuild_object_properties(&intern->std);
836:     }
837:
838:     if ((intern->var_flags & SPL_ARRAY_IS_SELF) {
839:         *is_temp = 0;
840:         return intern->std_properties;
841:     } else {
842:         HashTable *debug_info;
843:         *is_temp = 1;
844:
845:         debug_info = zend_new_array(zend_hash_num_elements(intern->std_properties) + 1);
846:         zend_hash_copy(debug_info, intern->std_properties, (copy_ctor_func_t) zval_add_ref);
847:
848:         storage = &intern->array;
849:         Z_TRY_ADDREF_P(storage);
850:
851:         base = Z_OBJ_HT_P(obj) == spl_handler_ArrayIterator
852:             ? spl_obj_ArrayIterator : spl_obj_ArrayObject;
853:         name = spl_get_private_prop_name(base, "storage", &storage) - 1;
854:         zend_symtable_update(debug_info, name, storage);
855:         zend_string_release(name);
856:
857:         return debug_info;
858:     }
859: } /* }}} */
860:
861: static HashTable *spl_array_get_gc(zval *obj, zval **gc_data, int *gc_data_count) /* {{{ */
862: {
863:     spl_array_object *intern = Z_SPLARRAY_P(obj);
864:
865:     *gc_data = &intern->array;
866:     *gc_data_count = 1;
867:     return zend_std_get_properties(obj);
868: } /* }}} */
869:
870: static zval *spl_array_read_property(zval *object, zval *member, int type, void **cache_slot, zval **rv) /* {{{ */
871: {
872:     spl_array_object *intern = Z_SPLARRAY_P(object);
873:
874:     if ((intern->var_flags & SPL_ARRAY_ARRAY_AS_PROPS) != 0
875:         && !std_object_handlers.has_property(object, member, 2, NULL)) {
876:         return spl_array_read_dimension(object, member, type, rv);
877:     }
878:
879:     return std_object_handlers.read_property(object, member, type, cache_slot, rv);
880: } /* }}} */
881:
882: static void spl_array_write_property(zval *object, zval *member, zval *value, void **cache_slot) /* {{{ */
883: {
884:     spl_array_object *intern = Z_SPLARRAY_P(object);
885:
886:     if ((intern->var_flags & SPL_ARRAY_ARRAY_AS_PROPS) != 0
887:         && !std_object_handlers.has_property(object, member, 2, NULL)) {
888:         spl_array_write_dimension(object, member, value);
889:         return;
890:     }
891:
892:     std_object_handlers.write_property(object, member, value, cache_slot);
893: } /* }}} */
894:
895: static zval *spl_array_get_property_ptr_ptr(zval *object, zval *member, int type, void **cache_slot) /* {{{ */
896: {
897:     spl_array_object *intern = Z_SPLARRAY_P(object);
898:
899:     if ((intern->var_flags & SPL_ARRAY_ARRAY_AS_PROPS) != 0
900:         && !std_object_handlers.has_property(object, member, 2, NULL)) {
901:         /* If object has offset() or offset(), then fallback to read_property,
902:            * which will call offsetGet(). */
903:         if ((intern->flags & SPL_OFFSET_GET) {
904:             return NULL;
905:         }
906:         return spl_array_get_dimension_ptr(1, intern, member, type);
907:     }
908:     return std_object_handlers.get_property_ptr_ptr(object, member, type, cache_slot);
909: } /* }}} */
910:
911: static int spl_array_has_property(zval *object, zval *member, int has_set_exists, void **cache_slot) /* {{{ */
912: {
913:     spl_array_object *intern = Z_SPLARRAY_P(object);
914:
915:     if ((intern->var_flags & SPL_ARRAY_ARRAY_AS_PROPS) != 0
916:         && !std_object_handlers.has_property(object, member, 2, NULL)) {
917:         return spl_array_has_dimension(object, member, has_set_exists);
918:     }
919:     return std_object_handlers.has_property(object, member, has_set_exists, cache_slot);
920: } /* }}} */
921:
922: static void spl_array_unset_property(zval *object, zval *member, void **cache_slot) /* {{{ */
923: {
924:     spl_array_object *intern = Z_SPLARRAY_P(object);
925:
926:     if ((intern->var_flags & SPL_ARRAY_ARRAY_AS_PROPS) != 0
927:         && !std_object_handlers.has_property(object, member, 2, NULL)) {
928:         spl_array_unset_dimension(object, member);
929:         return;
930:     }
931:     std_object_handlers.unset_property(object, member, cache_slot);
932: } /* }}} */
933:
934: static int spl_array_compare_objects(zval *o1, zval *o2) /* {{{ */
935: {
936:     HashTable *ht1,
937:               *ht2;
938:     spl_array_object *intern1,
939:                     *intern2;
940:     int result = 0;
941:
942:     intern1 = Z_SPLARRAY_P(o1);
943:     intern2 = Z_SPLARRAY_P(o2);
944:     ht1 = spl_array_get_hash_table(intern1);
945:     ht2 = spl_array_get_hash_table(intern2);
946:
947:     result = zend_compare_symbol_tables(ht1, ht2);
948:     /* If we just compared std_properties, don't do it again */
949:     if (result == 0 &&
950:         (!ht1 == intern1->std_properties && ht2 == intern2->std_properties)) {
951:         result = std_object_handlers.compare_objects(o1, o2);
952:     }
953:     return result;
954: } /* }}} */
955:
956: static int spl_array_skip_protected(spl_array_object *intern, HashTable *aht) /* {{{ */
957: {
958:     zend_string *attr_key;
959:     zend_ulong num_key;
960:     zval *data;
961:
962:     if (spl_array_is_object(intern) {
963:         uint32_t *pos_ptr = spl_array_get_pos_ptr(aht, intern);
964:
965:         do {
966:             if (zend_hash_get_current_key_ex(aht, attr_key, num_key, pos_ptr) == HASH_KEY_IS_STRING) {
967:                 data = zend_hash_get_current_data_ex(aht, pos_ptr);
968:                 if (data && Z_TYPE_P(data) == IS_INDIRECT) &&
969:                     Z_TYPE_P(*data) == Z_INDIRECT_P(data) == IS_UNDEF) {
970:                     /* skip */
971:                 } else if (!ZSTR_LEN(string_key) || ZSTR_VAL(string_key)[0]) {
972:                     return SUCCESS;
973:                 }
974:             } else {
975:                 return SUCCESS;
976:             }
977:             if (zend_hash_has_more_elements_ex(aht, pos_ptr) != SUCCESS) {
978:                 return FAILURE;
979:             }
980:             zend_hash_move_forward_ex(aht, pos_ptr);
981:         } while (1);
982:         return FAILURE;
983:     } /* }}} */
984:
985:     static int spl_array_next_ex(spl_array_object *intern, HashTable *aht) /* {{{ */
986:     {
987:         uint32_t *pos_ptr = spl_array_get_pos_ptr(aht, intern);
988:
989:         zend_hash_move_forward_ex(aht, pos_ptr);
990:         if (spl_array_is_object(intern) {
991:             return spl_array_skip_protected(intern, aht);
992:         } else {
993:             return zend_hash_has_more_elements_ex(aht, pos_ptr);
994:         }
995:     } /* }}} */
996:
997:     static int spl_array_next(spl_array_object *intern) /* {{{ */
998:     {
999:         HashTable *aht = spl_array_get_hash_table(intern);
1000:
1001:         return spl_array_next_ex(intern, aht);
1002:     } /* }}} */
1003:
1004:     static void spl_array_it_dtor(zend_object_iterator *iter) /* {{{ */
1005:     {
1006:         zend_user_it_invalidate_current(iter);
1007:         zval_ptr_dtor(&iter->data);
1008:     }
1009:     /* }}} */
1010:
1011:     static int spl_array_it_valid(zend_object_iterator *iter) /* {{{ */
1012:     {
1013:         spl_array_object *object = Z_SPLARRAY_P(iter->data);
1014:         HashTable *aht = spl_array_get_hash_table(object);
1015:
1016:         if (object->var_flags & SPL_ARRAY_OVERLOADED_VALID) {
1017:             return zend_user_it_valid(iter);
1018:         } else {
1019:             if (spl_array_object_verify_pos_ex(object, aht, "ArrayIterator::valid()") == FAILURE) {
1020:                 return FAILURE;
1021:             }
1022:             return zend_hash_has_more_elements_ex(aht, spl_array_get_pos_ptr(aht, object));
1023:         }
1024:     }
1025:     /* }}} */
1026:
1027:     static zval *spl_array_it_get_current_data(zend_object_iterator *iter) /* {{{ */
1028:     {
1029:         spl_array_object *object = Z_SPLARRAY_P(iter->data);
1030:         HashTable *aht = spl_array_get_hash_table(object);
1031:
1032:         if (object->var_flags & SPL_ARRAY_OVERLOADED_CURRENT) {
1033:             return zend_user_it_get_current_data(iter);
1034:         } else {
1035:             zval *data = zend_hash_get_current_data_ex(aht, spl_array_get_pos_ptr(aht, object));
1036:             if (Z_TYPE_P(data) == IS_INDIRECT) {
1037:                 data = Z_INDIRECT_P(data);
1038:             }
1039:             return data;
1040:         }
1041:     } /* }}} */
1042:
1043:     static void spl_array_it_get_current_key(zend_object_iterator *iter, zval **key) /* {{{ */
1044:     {
1045:         spl_array_object *object = Z_SPLARRAY_P(iter->data);
1046:         HashTable *aht = spl_array_get_hash_table(object);
1047:
1048:         if (object->var_flags & SPL_ARRAY_OVERLOADED_KEY) {
1049:             zend_user_it_get_current_key(iter, key);
1050:         } else {
1051:             if (spl_array_object_verify_pos_ex(object, aht, "ArrayIterator::current()") == FAILURE) {
1052:                 return NULL;
1053:             }
1054:             zend_hash_get_current_key_zval_ex(aht, key, spl_array_get_pos_ptr(aht, object));
1055:         }
1056:     } /* }}} */
1057:
1058:     static void spl_array_it_move_forward(zend_object_iterator *iter) /* {{{ */
1059:     {
1060:         spl_array_object *object = Z_SPLARRAY_P(iter->data);
1061:         HashTable *aht = spl_array_get_hash_table(object);
1062:
1063:         if (object->var_flags & SPL_ARRAY_OVERLOADED_NEXT) {
1064:             zend_user_it_move_forward(iter);
1065:         } else {
1066:             zend_user_it_invalidate_current(iter);
1067:             if (!aht) {
1068:                 php_error_docref(NULL, E_NOTICE, "ArrayIterator::current(): Array was modified outside object and is no longer an array");
1069:                 return;
1070:             }
1071:             spl_array_next_ex(object, aht);
1072:         }
1073:     } /* }}} */
1074:
1075:     static void spl_array_it_rewind(spl_array_object *intern) /* {{{ */
1076:     {
1077:         HashTable *aht = spl_array_get_hash_table(intern);
1078:
1079:         if (!aht) {
1080:             php_error_docref(NULL, E_NOTICE, "ArrayIterator::rewind(): Array was modified outside object and is no longer an array");
1081:             return;
1082:         }
1083:         if (intern->ht_iter == (uint32_t)-1) {
1084:             spl_array_get_pos_ptr(aht, intern);
1085:         } else {
1086:             zend_hash_internal_pointer_reset_ex(aht, spl_array_get_pos_ptr(aht, intern));
1087:             spl_array_skip_protected(intern, aht);
1088:         }
1089:     } /* }}} */
1090:
1091:     static void spl_array_it_rewind(zend_object_iterator *iter) /* {{{ */
1092:     {
1093:         spl_array_object *object = Z_SPLARRAY_P(iter->data);
1094:         HashTable *aht = spl_array_get_hash_table(object);
1095:
1096:         if (object->var_flags & SPL_ARRAY_OVERLOADED_REWIND) {
1097:             zend_user_it_rewind(iter);
1098:         } else {
1099:             zend_user_it_invalidate_current(iter);
1100:             spl_array_rewind(object);
1101:         }
1102:     } /* }}} */
1103:
1104:     static void spl_array_set_array(zval *object, spl_array_object *intern, zval *array, zend_ulong ar_flags, int just_array) {
1105:         if (Z_TYPE_P(array) != IS_ARRAY) {
1106:             zend_throw_exception(spl_ce_invalid_argument_exception, "Passed variable is not an array or object", 0);
1107:             return;
1108:         }
1109:
1110:         if (Z_TYPE_P(array) == IS_ARRAY) {
1111:             zval_ptr_dtor(&intern->array);
1112:             if (Z_REFCOUNT_P(array) == 1) {
1113:                 ZVAL_COPY(&intern->array, array);
1114:             } else {
1115:                 //??? TODO: try to avoid array duplication
1116:                 ZVAL_ARR(&intern->array, zend_array_dup(Z_ARR_P(array)));
1117:             }
1118:         } else {
1119:

```

```

1129:         IF (Z_OBJ_HT_P(array) == spl_handler_arrayobject || Z_OBJ_HT_P(array) == spl_handler_arrayiterator)
1130:             zval_ptr_dtor(&intern->array);
1131:         IF (!zval_array())
1132:             spl_array_object(&other = Z_SPLARRAY_P(array));
1133:             ar_flags = &other->ar_flags & Z_SPLARRAY_INT_MASK;
1134:
1135:         IF (Z_OBJ_P(object) == Z_OBJ_Z(array)) {
1136:             ar_flags |= SPL_ARRAY_IS_SELF;
1137:             ZVAL_UNDEF(&intern->array);
1138:         } else {
1139:             ar_flags |= SPL_ARRAY_USE_OTHER;
1140:             ZVAL_COPY(&intern->array, array);
1141:         }
1142:     } else {
1143:         zend_object_get_properties_t handler = Z_OBJ_HANDLER_P(array, get_properties);
1144:         IF (handler != std_object_handlers.get_properties()) {
1145:             zend_throw_exception_ex(spl_ce_invalidargumentexception, 0,
1146:                 "overloaded object of type %s is not compatible with %s",
1147:                 ZSTR_VAL(Z_OBJ_CK_P(array)->name), ZSTR_VAL(intern->std.ce->name));
1148:         }
1149:     }
1150:     zval_ptr_dtor(&intern->array);
1151:     ZVAL_COPY(&intern->array, array);
1152: }
1153: }
1154:
1155: intern->ar_flags = "SPL_ARRAY_IS_SELF" & "SPL_ARRAY_USE_OTHER";
1156: intern->ar_flags |= ar_flags;
1157: intern->ht_iter = (uint32_t)-1;
1158: }
1159: /* }}} */
1160:
1161: /* Iterator handler table */
1162: static const zend_object_iterator_funcs spl_array_it_funcs = {
1163:     spl_array_it_dtor,
1164:     spl_array_it_valid,
1165:     spl_array_it_get_current_data,
1166:     spl_array_it_get_current_key,
1167:     spl_array_it_move_forward,
1168:     spl_array_it_rewind,
1169:     NULL
1170: };
1171:
1172: zend_object_iterator *spl_array_get_iterator(zend_class_entry *ce, zval *object, int by_ref) /* {{{ */
1173: {
1174:     zend_object_iterator *iterator;
1175:     zend_array_object *array_object = Z_SPLARRAY_P(object);
1176:
1177:     IF (by_ref & (array_object->ar_flags & SPL_ARRAY_OVERLOADED_CURRENT)) {
1178:         zend_throw_exception(spl_ce_runtimeexception, "An iterator cannot be used with foreach by reference", 0);
1179:         return NULL;
1180:     }
1181:
1182:     iterator = emalloc(sizeof(zend_object_iterator));
1183:
1184:     zend_iterator_init(&iterator->it);
1185:
1186:     ZVAL_COPY(&iterator->it.data, object);
1187:     iterator->it.funcs = spl_array_it_funcs;
1188:     iterator->ce = ce;
1189:     ZVAL_UNDEF(&iterator->value);
1190:
1191:     return iterator->it;
1192: }
1193: /* }}} */
1194:
1195: /* {{{ proto void ArrayObject::__construct([array|object ar = array() |, int flags = 0 |, string iterator_class = null])
1196:    Constructs a new array iterator from an array or object. */
1197: SPL_METHOD(Array, __construct)
1198: {
1199:     zval *object = getThis();
1200:     spl_array_object *intern;
1201:     zval *array;
1202:     zend_long ar_flags = 0;
1203:     zend_class_entry *ce_get_iterator = spl_ce_iterator;
1204:
1205:     IF (ZEND_NUM_ARGS() == 0) {
1206:         return; /* nothing to do */
1207:     }
1208:
1209:     IF (zend_parse_parameters_throw(ZEND_NUM_ARGS(), "m|l", &array, &ar_flags, &ce_get_iterator) == FAILURE) {
1210:         return;
1211:     }
1212:
1213:     intern = Z_SPLARRAY_P(object);
1214:
1215:     IF (ZEND_NUM_ARGS() > 2) {
1216:         intern->ce_get_iterator = ce_get_iterator;
1217:     }
1218:
1219:     ar_flags |= "SPL_ARRAY_INT_MASK";
1220:
1221:     spl_array_set_array(object, intern, array, ar_flags, ZEND_NUM_ARGS() == 1);
1222: }
1223: /* }}} */
1224:
1225: /* {{{ proto void ArrayIterator::__construct([array|object ar = array() |, int flags = 0])
1226:    Constructs a new array iterator from an array or object. */
1227: SPL_METHOD(ArrayIterator, __construct)
1228: {
1229:     zval *object = getThis();
1230:     spl_array_object *intern;
1231:     zval *array;
1232:     zend_long ar_flags = 0;
1233:
1234:     IF (ZEND_NUM_ARGS() == 0) {
1235:         return; /* nothing to do */
1236:     }
1237:
1238:     IF (zend_parse_parameters_throw(ZEND_NUM_ARGS(), "m|l", &array, &ar_flags) == FAILURE) {
1239:         return;
1240:     }
1241:
1242:     intern = Z_SPLARRAY_P(object);
1243:
1244:     ar_flags |= "SPL_ARRAY_INT_MASK";
1245:
1246:     spl_array_set_array(object, intern, array, ar_flags, ZEND_NUM_ARGS() == 1);
1247: }
1248: /* }}} */
1249:
1250: /* {{{ proto void ArrayObject::setIteratorClass(string iterator_class)
1251:    Set the class used in getIterator. */
1252: SPL_METHOD(Array, setIteratorClass)
1253: {
1254:     zval *object = getThis();
1255:     spl_array_object *intern = Z_SPLARRAY_P(object);
1256:     zend_class_entry *ce_get_iterator = spl_ce_iterator;
1257:
1258:     intern->PARAMETERS->start(1, 1)
1259:     Z_PARAM_CLASS(ce_get_iterator)
1260:     ZEND_PARSE_PARAMETERS_END();
1261:
1262:     intern->ce_get_iterator = ce_get_iterator;
1263: }
1264: /* }}} */
1265:
1266: /* {{{ proto string ArrayObject::getIteratorClass()
1267:    Get the class used in getIterator. */
1268: SPL_METHOD(Array, getIteratorClass)
1269: {
1270:     zval *object = getThis();
1271:     spl_array_object *intern = Z_SPLARRAY_P(object);
1272:
1273:     IF (zend_parse_parameters_none() == FAILURE) {
1274:         return;
1275:     }
1276:
1277:     zend_string_addref(&intern->ce_get_iterator->name);
1278:     RETURN_STR(&intern->ce_get_iterator->name);
1279: }
1280: /* }}} */
1281:
1282: /* {{{ proto int ArrayObject::getFlags()
1283:    Get flags */
1284: SPL_METHOD(Array, getFlags)
1285: {
1286:     zval *object = getThis();
1287:     spl_array_object *intern = Z_SPLARRAY_P(object);
1288:
1289:     IF (zend_parse_parameters_none() == FAILURE) {
1290:         return;
1291:     }
1292:
1293:     RETURN_LONG(intern->ar_flags & SPL_ARRAY_INT_MASK);
1294: }
1295: /* }}} */
1296:
1297: /* {{{ proto void ArrayObject::setFlags(int flags)
1298:    Set flags */
1299: SPL_METHOD(Array, setFlags)
1300: {
1301:     zval *object = getThis();
1302:     spl_array_object *intern = Z_SPLARRAY_P(object);
1303:     zend_long ar_flags = 0;
1304:
1305:     IF (zend_parse_parameters(ZEND_NUM_ARGS(), "l", &ar_flags) == FAILURE) {
1306:         return;
1307:     }
1308:
1309:     intern->ar_flags = (intern->ar_flags & SPL_ARRAY_INT_MASK) | (ar_flags & "SPL_ARRAY_INT_MASK");
1310: }
1311: /* }}} */
1312:
1313: /* {{{ proto ArrayObject ArrayObject::exchangeArray(ArrayObject ar = array())
1314:    Replaces the referenced array or object with a new one and return the old one (right now copy - to be changed) */
1315: SPL_METHOD(Array, exchangeArray)
1316: {

```

```

3119:     eval *object = getThis(), *array;
3120:     spl_array_object *intern = _SPLARRAY_P(object);
3121: }
3122:
3123:
3124: if (zend_parse_parameters(ZEND_NUM_ARGS() == 1, "s", &array) == FAILURE) {
3125:     return;
3126: }
3127:
3128:
3129: RETVAL_ARR(zend_array_dup(spl_array_get_hash_table(intern)));
3130: spl_array_set_array(object, intern, array, 0);
3131: }
3132: /* }}} */
3133:
3134: /* {{{ proto ArrayIterator ArrayObject::getIterator()
3135:  * Create a new iterator from a ArrayObject instance */
3136: SPL_METHOD(Array, getIterator)
3137: {
3138:     eval *object = getThis();
3139:     spl_array_object *intern = _SPLARRAY_P(object);
3140:     HashTable *ah = spl_array_get_hash_table(intern);
3141:
3142: if (zend_parse_parameters_none() == FAILURE) {
3143:     return;
3144: }
3145:
3146: if (ah) {
3147:     php_error_docref(NULL, E_NOTICE, "Array was modified outside object and is no longer an array");
3148:     return;
3149: }
3150:
3151: ZVAL_OBJ(&return_value, spl_array_object_new_ex(intern->obj->get_iterator(), object, 0));
3152: }
3153: /* }}} */
3154:
3155: /* {{{ proto void ArrayIterator::rewind()
3156:  * Rewind array back to the start */
3157: SPL_METHOD(Array, rewind)
3158: {
3159:     eval *object = getThis();
3160:     spl_array_object *intern = _SPLARRAY_P(object);
3161:
3162: if (zend_parse_parameters_none() == FAILURE) {
3163:     return;
3164: }
3165:
3166: spl_array_rewind(intern);
3167: }
3168: /* }}} */
3169:
3170: /* {{{ proto void ArrayIterator::seek(int $position)
3171:  * Seek to position. */
3172: SPL_METHOD(Array, seek)
3173: {
3174:     zend_long opos, position;
3175:     eval *object = getThis();
3176:     spl_array_object *intern = _SPLARRAY_P(object);
3177:     HashTable *ah = spl_array_get_hash_table(intern);
3178:     int result;
3179:
3180: if (zend_parse_parameters(ZEND_NUM_ARGS() == 1, "i", &position) == FAILURE) {
3181:     return;
3182: }
3183:
3184: if (ah) {
3185:     php_error_docref(NULL, E_NOTICE, "Array was modified outside object and is no longer an array");
3186:     return;
3187: }
3188:
3189: opos = position;
3190:
3191: if (position == 0) { /* negative values are not supported */
3192:     spl_array_rewind(intern);
3193:     result = SUCCESS;
3194:
3195:     while (position-- > 0 && (result = spl_array_next(intern)) == SUCCESS;
3196:
3197: if (result == SUCCESS && zend_hash_has_more_elements_ex(ah, spl_array_get_pos_ptr(ah, intern)) == SUCCESS) {
3198:     return; /* ok */
3199: }
3200: }
3201:
3202: zend_throw_exception_ex(spl_ce_OutOfBoundsException, 0, "Seek position '%d' is out of range", opos);
3203: }
3204:
3205: static int spl_array_object_count_elements_helper(spl_array_object *intern, zend_long *count) /* {{{ */
3206: {
3207:     HashTable *ah = spl_array_get_hash_table(intern);
3208:     HashPosition pos, *pos_ptr;
3209:
3210: if (ah) {
3211:     zend_throw_exception(NULL, E_NOTICE, "Array was modified outside object and is no longer an array");
3212:     *count = 0;
3213:     return FAILURE;
3214: }
3215:
3216: if (spl_array_is_object(intern))
3217:     /* We need to store the 'pos' since we'll modify it in the functions
3218:      * we're going to call and which do not support 'pos' as parameter. */
3219:     pos_ptr = spl_array_get_pos_ptr(ah, intern);
3220:     pos = *pos_ptr;
3221:     *count = 0;
3222:     zend_array_rewind(intern);
3223:     while ((pos_ptr = HT_INVALID_IDX & spl_array_next(intern) == SUCCESS) {
3224:         (*count)++;
3225:     }
3226:     *pos_ptr = pos;
3227:     return SUCCESS;
3228: } else {
3229:     *count = zend_hash_num_elements(ah);
3230:     return SUCCESS;
3231: }
3232: }
3233: /* }}} */
3234:
3235: static int spl_array_object_count_elements(spl_array_object *intern, zend_long *count) /* {{{ */
3236: {
3237:     spl_array_object *intern = _SPLARRAY_P(object);
3238:
3239: if (intern->fptr->count) {
3240:     spl_rval;
3241:     zend_call_method_with_0_params(object, intern->std_obj, &intern->fptr->count, "count", &rv);
3242:     if (IS_TYPE_IV == IS_INTEGER)
3243:         *count = zend_get_long(rv);
3244:         zend_get_rsrc(rv);
3245:         return SUCCESS;
3246:     }
3247:     *count = 0;
3248:     return FAILURE;
3249: }
3250: return spl_array_object_count_elements_helper(intern, count);
3251: }
3252: /* }}} */
3253:
3254: static int spl_array_object_count(spl_array_object *intern)
3255: {
3256:     proto int ArrayObject::count()
3257:     proto int ArrayIterator::count()
3258:     return the number of elements in the Iterator. */
3259: SPL_METHOD(Array, count)
3260: {
3261:     zend_long count;
3262:     spl_array_object *intern = _SPLARRAY_P(getThis());
3263:
3264: if (zend_parse_parameters_none() == FAILURE) {
3265:     return;
3266: }
3267:
3268: spl_array_object_count_elements_helper(intern, &count);
3269:
3270: RETURN_LONG(count);
3271: }
3272: /* }}} */
3273:
3274: static void spl_array_method(INTERNAL_FUNCTION_PARAMETERS, char *fname, int fname_len, int use_arg) /* {{{ */
3275: {
3276:     call_user_function_ex(EG(function_table), NULL, &function_name, return_value, 1, params, 1, NULL);
3277:     zend_call_method_with_0_params(object, intern->std_obj, &intern->fptr->count, "count", &rv);
3278:     HashTable *ah = spl_array_get_hash_table(intern);
3279:     eval function_name, params[2], *arg = NULL;
3280:
3281: ZVAL_STRING(&function_name, fname, fname_len);
3282:
3283: EVAL_NRM_INIT_FUNC(params[0]);
3284: EVAL_ARR_INIT_FUNC(params[0], ah);
3285: GC_ADDREF(ah);
3286:
3287: if (!use_arg) {
3288:     intern->nApplyCount++;
3289:     call_user_function_ex(EG(function_table), NULL, &function_name, return_value, 1, params, 1, NULL);
3290:     return;
3291: } else if (use_arg == SPL_ARRAY_METHOD_MAY_USER_ARG) {
3292:     if (zend_parse_parameters_ex(ZEND_PARSE_PARAMS_QUIET, ZEND_NUM_ARGS() == 1, "s", &arg) == FAILURE) {
3293:         zend_throw_exception(spl_ce_RuntimeException, "Function expects one argument at most", 0);
3294:         goto exit;
3295:     }
3296:     if (arg) {
3297:         EVAL_COPY_VALUE(&params[1], arg);
3298:     }
3299:     return;
3300: } else if (use_arg == SPL_ARRAY_METHOD_MAY_USER_ARG_2) {
3301:     call_user_function_ex(EG(function_table), NULL, &function_name, return_value, 2, params, 1, NULL);
3302:     return;
3303: } else if (ZEND_NUM_ARGS() > 1) {
3304:     zend_parse_parameters_ex(ZEND_PARSE_PARAMS_QUIET, ZEND_NUM_ARGS() == 2, "s", &arg) == FAILURE) {
3305:         zend_throw_exception(spl_ce_RuntimeException, "Function expects exactly one argument", 0);
3306:         goto exit;
3307:     }
3308:     EVAL_COPY_VALUE(&params[1], arg);
3309:     return;
3310: } else {
3311:     call_user_function_ex(EG(function_table), NULL, &function_name, return_value, 2, params, 1, NULL);
3312:     return;
3313: }
3314: }

```

```
1505:
1506: emit:
1507: {
1508:     HashTable *new_ht = _Z_ARRVAL_P(_Z_NEWVAL(params[0]));
1509:     if (abt != new_ht) {
1510:         spl_array_replace_hash_table(intern, new_ht);
1511:     } else {
1512:         GC_DELREF(abt);
1513:     }
1514:     _Z_ADDREF(_Z_STR(function_name));
1515:     zend_string_free(_Z_STR(function_name));
1516: }
1517: /* }}} */
1518:
1519: #define SPL_ARRAY_METHOD(cname, fname, use_arg) \
1520: SPL_METHOD(cname, fname) { \
1521:     /* \
1522:     spl_array_method(INTERNAL_FUNCTION_PARAM_PASSTHRU, #fname, sizeof(#fname)-1, use_arg); \
1523:     */ \
1524:
1525: /* {{{ proto int ArrayObject::asort([int $sort_flags = SORT_REGULAR]) \
1526:    proto int ArrayIterator::asort([int $sort_flags = SORT_REGULAR]) \
1527:    Sort the entries by values. */ \
1528: SPL_ARRAY_METHOD(Array, asort, SPL_ARRAY_METHOD_MAY_USER_ARG) /* }}} */
1529:
1530: /* {{{ proto int ArrayObject::ksort([int $sort_flags = SORT_REGULAR]) \
1531:    proto int ArrayIterator::ksort([int $sort_flags = SORT_REGULAR]) \
1532:    Sort the entries by key. */ \
1533: SPL_ARRAY_METHOD(Array, ksort, SPL_ARRAY_METHOD_MAY_USER_ARG) /* }}} */
1534:
1535: /* {{{ proto int ArrayObject::uasort(callback cmp_function) \
1536:    proto int ArrayIterator::uasort(callback cmp_function) \
1537:    Sort the entries by values using defined function. */ \
1538: SPL_ARRAY_METHOD(Array, uasort, SPL_ARRAY_METHOD_MAY_USER_ARG) /* }}} */
1539:
1540: /* {{{ proto int ArrayObject::uksort(callback cmp_function) \
1541:    proto int ArrayIterator::uksort(callback cmp_function) \
1542:    Sort the entries by key using user defined function. */ \
1543: SPL_ARRAY_METHOD(Array, uksort, SPL_ARRAY_METHOD_MAY_USER_ARG) /* }}} */
1544:
1545: /* {{{ proto int ArrayObject::natsort() \
1546:    proto int ArrayIterator::natsort() \
1547:    Sort the entries by values using "natural order" algorithm. */ \
1548: SPL_ARRAY_METHOD(Array, natsort, SPL_ARRAY_METHOD_MAY_USER_ARG) /* }}} */
1549:
1550: /* {{{ proto int ArrayObject::natcasesort() \
1551:    proto int ArrayIterator::natcasesort() \
1552:    Sort the entries by key using case insensitive "natural order" algorithm. */ \
1553: SPL_ARRAY_METHOD(Array, natcasesort, SPL_ARRAY_METHOD_MAY_USER_ARG) /* }}} */
1554:
1555: /* {{{ proto mixed NULL ArrayIterator::current() \
1556:    Return current array entry */ \
1557: SPL_METHOD(Array, current)
1558: {
1559:     zval *object = getThis();
1560:     spl_array_object *intern = _Z_SPLARRAY_P(object);
1561:     zval *entry;
1562:     HashTable *abt = spl_array_get_hash_table(intern);
1563:
1564:     if (zend_parse_parameters_none() == FAILURE) {
1565:         return;
1566:     }
1567:
1568:     if (spl_array_object_verify_pos(intern, abt) == FAILURE) {
1569:         return;
1570:     }
1571:
1572:     if (entry = zend_hash_get_current_data_ex(abt, spl_array_get_pos_ptr(abt, intern)) == NULL) {
1573:         return;
1574:     }
1575:     if (_Z_TYPE_P(entry) == IS_INDIRECT) {
1576:         entry = _Z_INDIRECT_P(entry);
1577:     }
1578:     if (_Z_TYPE_P(entry) == IS_UNDEF) {
1579:         return;
1580:     }
1581:     ZVAL_DEREF(entry);
1582:     ZVAL_COPY(return_value, entry);
1583: }
1584: /* }}} */
1585:
1586: /* {{{ proto mixed NULL ArrayIterator::key() \
1587:    Return current array key */ \
1588: SPL_METHOD(Array, key)
1589: {
1590:     if (zend_parse_parameters_none() == FAILURE) {
1591:         return;
1592:     }
1593:
1594:     spl_array_iterator_key(getThis(), return_value);
1595:     /* }}} */
1596:
1597: void spl_array_iterator_key(zval *object, zval *return_value) /* {{{ */
1598: {
1599:     spl_array_object *intern = _Z_SPLARRAY_P(object);
1600:     HashTable *abt = spl_array_get_hash_table(intern);
1601:
1602:     if (spl_array_object_verify_pos(intern, abt) == FAILURE) {
1603:         return;
1604:     }
1605:
1606:     zend_hash_get_current_key_val_ex(abt, return_value, spl_array_get_pos_ptr(abt, intern));
1607: }
1608: /* }}} */
1609:
1610: /* {{{ proto void ArrayIterator::next() \
1611:    Move to next entry */ \
1612: SPL_METHOD(Array, next)
1613: {
1614:     zval *object = getThis();
1615:     spl_array_object *intern = _Z_SPLARRAY_P(object);
1616:     HashTable *abt = spl_array_get_hash_table(intern);
1617:
1618:     if (zend_parse_parameters_none() == FAILURE) {
1619:         return;
1620:     }
1621:
1622:     if (spl_array_object_verify_pos(intern, abt) == FAILURE) {
1623:         return;
1624:     }
1625:
1626:     spl_array_next_ex(intern, abt);
1627: }
1628: /* }}} */
1629:
1630: /* {{{ proto bool ArrayIterator::valid() \
1631:    Check whether array contains more entries */ \
1632: SPL_METHOD(Array, valid)
1633: {
1634:     zval *object = getThis();
1635:     spl_array_object *intern = _Z_SPLARRAY_P(object);
1636:     HashTable *abt = spl_array_get_hash_table(intern);
1637:
1638:     if (zend_parse_parameters_none() == FAILURE) {
1639:         return;
1640:     }
1641:
1642:     if (spl_array_object_verify_pos(intern, abt) == FAILURE) {
1643:         RETURN_FALSE;
1644:     } else {
1645:         RETURN_BOOL(zend_hash_has_more_elements_ex(abt, spl_array_get_pos_ptr(abt, intern)) == SUCCESS);
1646:     }
1647: }
1648: /* }}} */
1649:
1650: /* {{{ proto bool RecursiveArrayIterator::hasChildren() \
1651:    Check whether current element has children (e.g. is an array) */ \
1652: SPL_METHOD(Array, hasChildren)
1653: {
1654:     zval *object = getThis(), *entry;
1655:     spl_array_object *intern = _Z_SPLARRAY_P(object);
1656:     HashTable *abt = spl_array_get_hash_table(intern);
1657:
1658:     if (zend_parse_parameters_none() == FAILURE) {
1659:         return;
1660:     }
1661:
1662:     if (spl_array_object_verify_pos(intern, abt) == FAILURE) {
1663:         RETURN_FALSE;
1664:     }
1665:
1666:     if (entry = zend_hash_get_current_data_ex(abt, spl_array_get_pos_ptr(abt, intern)) == NULL) {
1667:         RETURN_FALSE;
1668:     }
1669:
1670:     if (_Z_TYPE_P(entry) == IS_INDIRECT) {
1671:         entry = _Z_INDIRECT_P(entry);
1672:     }
1673:
1674:     ZVAL_DEREF(entry);
1675:     RETURN_BOOL(_Z_TYPE_P(entry) == IS_ARRAY || (_Z_TYPE_P(entry) == IS_OBJECT && (intern->var_flags & SPL_ARRAY_CHILD_ARRAYS_ONLY) == 0));
1676: }
1677: /* }}} */
1678:
1679: /* {{{ proto object RecursiveArrayIterator::getChildren() \
1680:    Create a sub iterator for the current element (same class as $this) */ \
1681: SPL_METHOD(Array, getChildren)
1682: {
1683:     zval *object = getThis(), *entry, flags;
1684:     spl_array_object *intern = _Z_SPLARRAY_P(object);
1685:     HashTable *abt = spl_array_get_hash_table(intern);
1686:
1687:     if (zend_parse_parameters_none() == FAILURE) {
1688:         return;
1689:     }
1690:
1691:     if (spl_array_object_verify_pos(intern, abt) == FAILURE) {
1692:         return;
1693:     }
1694: }
```

```
1695: }
1696:
1697: if (entry = zend_hash_get_current_data_ex(abt, spl_array_get_pos_ptr(abt, intern)) == NULL) {
1698:     return;
1699: }
1700:
1701: if (_Z_TYPE_P(entry) == IS_INDIRECT) {
1702:     entry = _Z_INDIRECT_P(entry);
1703: }
1704:
1705: ZVAL_DEREF(entry);
1706:
1707: if (_Z_TYPE_P(entry) == IS_OBJECT) {
1708:     if ((intern->var_flags & SPL_ARRAY_CHILD_ARRAYS_ONLY) != 0) {
1709:         return;
1710:     }
1711:     if (instanceof_function(Z_OBJCE_P(entry), _Z_OBJCE_P(getThis())) {
1712:         EVAL_OBJ(return_value, _Z_OBJ_P(entry));
1713:     }
1714:     ZVAL_COPY(return_value, entry);
1715: }
1716:
1717: /* }}} */
1718:
1719: /* {{{ proto string ArrayObject::serialize() \
1720:    Serialize the object */ \
1721: SPL_METHOD(Array, serialize)
1722: {
1723:     zval *object = getThis();
1724:     spl_array_object *intern = _Z_SPLARRAY_P(object);
1725:     HashTable *abt = spl_array_get_hash_table(intern);
1726:     zval members, flags;
1727:     php_serialize_data_t var_hash;
1728:     smart_str buf = {0};
1729:
1730:     if (zend_parse_parameters_none() == FAILURE) {
1731:         return;
1732:     }
1733:
1734:     if (abt) {
1735:         php_error_docref(NULL, E_NOTICE, "Array was modified outside object and is no longer an array");
1736:         return;
1737:     }
1738:
1739:     PHP_VAR_SERIALIZE_INIT(var_hash);
1740:
1741:     ZVAL_LONG(&flags, (intern->var_flags & SPL_ARRAY_CLONE_MASK));
1742:
1743:     /* storage */
1744:     smart_str_appendl(&buf, "a:", 2);
1745:     php_var_serialize(&buf, &flags, &var_hash);
1746:
1747:     if ((intern->var_flags & SPL_ARRAY_IS_SELF) != 0) {
1748:         php_var_serialize(&buf, &intern->array, &var_hash);
1749:         smart_str_appendl(&buf, ";");
1750:     }
1751:
1752:     /* members */
1753:     smart_str_appendl(&buf, "m:", 2);
1754:     if ((intern->std.properties) != 0) {
1755:         rebuild_object_properties(&intern->std);
1756:     }
1757:
1758:     ZVAL_ARR(&members, &intern->std.properties);
1759:
1760:     php_var_serialize(&buf, &members, &var_hash); /* finishes the string */
1761:
1762:     /* done */
1763:     PHP_VAR_SERIALIZE_DESTROY(var_hash);
1764:
1765:     if (buf.s) {
1766:         RETURN_NEW_STR(buf.s);
1767:     }
1768: }
1769:
1770: RETURN_NULL();
1771: /* }}} */
1772:
1773: /* {{{ proto void ArrayObject::unserialize(string $serialized) \
1774:    * unserialize the object */ \
1775: */ \
1776: SPL_METHOD(Array, unserialize)
1777: {
1778:     zval *object = getThis();
1779:     spl_array_object *intern = _Z_SPLARRAY_P(object);
1780:
1781:     char *buf;
1782:     size_t buf_len;
1783:     const unsigned char *p;
1784:     php_unserialize_data_t var_hash;
1785:     zval *members, *flags, *array;
1786:     zend_long flags;
1787:
1788:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "s", &buf, &buf_len) == FAILURE) {
1789:         return;
1790:     }
1791:
1792:     if (buf_len == 0) {
1793:         return;
1794:     }
1795:
1796:     if ((intern->applyCount > 0) {
1797:         zend_error(E_WARNING, "Modification of ArrayObject during sorting is prohibited");
1798:         return;
1799:     }
1800:
1801:     /* storage */
1802:     s = p = (const unsigned char*)buf;
1803:     PHP_VAR_UNSERIALIZE_INIT(var_hash);
1804:
1805:     if ('p' != *p || **p != ':') {
1806:         goto outexcept;
1807:     }
1808:     *p;
1809:
1810:     flags = var_tmp_var(&var_hash);
1811:     if (php_var_unserialize(&flags, s + buf_len, &var_hash) || _Z_TYPE_P(flags) != IS_LONG) {
1812:         goto outexcept;
1813:     }
1814:
1815:     --p; /* for 'p' */
1816:     flags = _Z_LONG_P(flags);
1817:     /* flags needs to be verified and we also need to verify whether the next
1818:     * thing we get is 'r'. After that we require an 'a' or something else
1819:     * where 'a' stands for members and anything else should be an array. If
1820:     * neither 'a' or 'a' follows we have an error. */
1821:
1822:     if ('r' != *p) {
1823:         goto outexcept;
1824:     }
1825:     *p;
1826:
1827:     if (flags & SPL_ARRAY_IS_SELF) {
1828:         /* If IS_SELF is used, the flags are not followed by an array/object */
1829:         intern->var_flags &= "SPL_ARRAY_CLONE_MASK";
1830:         intern->var_flags |= flags & SPL_ARRAY_CLONE_MASK;
1831:         zval_get_ref(intern->array);
1832:         ZVAL_UNDEF(intern->array);
1833:     } else {
1834:         if ('p' != *p && 'p' != 'O' && 'p' != 'C' && 'p' != 'r') {
1835:             goto outexcept;
1836:         }
1837:
1838:         array = var_tmp_var(&var_hash);
1839:         if (php_var_unserialize(array, s + buf_len, &var_hash)
1840:             || (_Z_TYPE_P(array) != IS_ARRAY && _Z_TYPE_P(array) != IS_OBJECT)) {
1841:             goto outexcept;
1842:         }
1843:
1844:         intern->var_flags &= "SPL_ARRAY_CLONE_MASK";
1845:         intern->var_flags |= flags & SPL_ARRAY_CLONE_MASK;
1846:
1847:         if (_Z_TYPE_P(array) == IS_ARRAY) {
1848:             zval_get_ref(intern->array);
1849:             ZVAL_COPY(intern->array, array);
1850:         } else {
1851:             spl_array_set_array(object, intern, array, 0, 1);
1852:         }
1853:
1854:         if ('r' != *p) {
1855:             goto outexcept;
1856:         }
1857:         *p;
1858:
1859:         /* members */
1860:         if ('m' != *p || **p != ':') {
1861:             goto outexcept;
1862:         }
1863:         *p;
1864:
1865:         members = var_tmp_var(&var_hash);
1866:         if (php_var_unserialize(members, s + buf_len, &var_hash) || _Z_TYPE_P(members) != IS_ARRAY) {
1867:             goto outexcept;
1868:         }
1869:     }
1870:
1871:     /* copy members */
1872:     object_properties_load(intern->std, _Z_ARRVAL_P(members));
1873:
1874:     /* done reading Serialized */
1875:     PHP_VAR_UNSERIALIZE_DESTROY(var_hash);
1876:     return;
1877:
1878: outexcept:
1879:     PHP_VAR_UNSERIALIZE_DESTROY(var_hash);
1880:     zend_throw_exception(expl_ce, UnexpectedValueException, 0, "Error at offset " ZEND_LONG_FMT " of %d bytes", (zend_long)(char*)p - buf, buf_len);
1881: }
```

```
1881:     return;
1882:
1883: } /* }}} */
1884:
1885: /* {{{ arginfo and function table */
1886: ZEND_BEGIN_ARG_INFO_EX(arginfo_array___construct, 0, 0, 0)
1887:     ZEND_ARG_INFO(0, array)
1888:     ZEND_ARG_INFO(0, ar_flags)
1889:     ZEND_ARG_INFO(0, iterator_class)
1890: ZEND_END_ARG_INFO()
1891:
1892: /* ArrayIterator::__construct and ArrayObject::__construct have different signatures */
1893: ZEND_BEGIN_ARG_INFO_EX(arginfo_array_iterator___construct, 0, 0, 0)
1894:     ZEND_ARG_INFO(0, array)
1895:     ZEND_ARG_INFO(0, ar_flags)
1896: ZEND_END_ARG_INFO()
1897:
1898: ZEND_BEGIN_ARG_INFO_EX(arginfo_array_offsetGet, 0, 0, 1)
1899:     ZEND_ARG_INFO(0, index)
1900: ZEND_END_ARG_INFO()
1901:
1902: ZEND_BEGIN_ARG_INFO_EX(arginfo_array_offsetSet, 0, 0, 2)
1903:     ZEND_ARG_INFO(0, index)
1904:     ZEND_ARG_INFO(0, newval)
1905: ZEND_END_ARG_INFO()
1906:
1907: ZEND_BEGIN_ARG_INFO_EX(arginfo_array_append, 0)
1908:     ZEND_ARG_INFO(0, value)
1909: ZEND_END_ARG_INFO()
1910:
1911: ZEND_BEGIN_ARG_INFO_EX(arginfo_array_seek, 0)
1912:     ZEND_ARG_INFO(0, position)
1913: ZEND_END_ARG_INFO()
1914:
1915: ZEND_BEGIN_ARG_INFO_EX(arginfo_array_exchangeArray, 0)
1916:     ZEND_ARG_INFO(0, array)
1917: ZEND_END_ARG_INFO()
1918:
1919: ZEND_BEGIN_ARG_INFO_EX(arginfo_array_setFlags, 0)
1920:     ZEND_ARG_INFO(0, flags)
1921: ZEND_END_ARG_INFO()
1922:
1923: ZEND_BEGIN_ARG_INFO_EX(arginfo_array_getIteratorClass, 0)
1924:     ZEND_ARG_INFO(0, iteratorClass)
1925: ZEND_END_ARG_INFO()
1926:
1927: ZEND_BEGIN_ARG_INFO_EX(arginfo_array_uksort, 0)
1928:     ZEND_ARG_INFO(0, cmp_function)
1929: ZEND_END_ARG_INFO()
1930:
1931: ZEND_BEGIN_ARG_INFO_EX(arginfo_array_unserialize, 0)
1932:     ZEND_ARG_INFO(0, serialized)
1933: ZEND_END_ARG_INFO()
1934:
1935: ZEND_BEGIN_ARG_INFO_EX(arginfo_array_void, 0)
1936: ZEND_END_ARG_INFO()
1937:
1938: static const zend_function_entry spl_funcs_arrayObject[] = {
1939:     SPL_ME(Array, __construct, arginfo_array___construct, ZEND_ACC_PUBLIC)
1940:     SPL_ME(Array, offsetExists, arginfo_array_offsetGet, ZEND_ACC_PUBLIC)
1941:     SPL_ME(Array, offsetGet, arginfo_array_offsetGet, ZEND_ACC_PUBLIC)
1942:     SPL_ME(Array, offsetSet, arginfo_array_offsetSet, ZEND_ACC_PUBLIC)
1943:     SPL_ME(Array, offsetUnset, arginfo_array_offsetGet, ZEND_ACC_PUBLIC)
1944:     SPL_ME(Array, append, arginfo_array_append, ZEND_ACC_PUBLIC)
1945:     SPL_ME(Array, getArrayCopy, arginfo_array_void, ZEND_ACC_PUBLIC)
1946:     SPL_ME(Array, count, arginfo_array_void, ZEND_ACC_PUBLIC)
1947:     SPL_ME(Array, getFlags, arginfo_array_void, ZEND_ACC_PUBLIC)
1948:     SPL_ME(Array, setFlags, arginfo_array_setFlags, ZEND_ACC_PUBLIC)
1949:     SPL_ME(Array, asort, arginfo_array_void, ZEND_ACC_PUBLIC)
1950:     SPL_ME(Array, usort, arginfo_array_void, ZEND_ACC_PUBLIC)
1951:     SPL_ME(Array, uasort, arginfo_array_uksort, ZEND_ACC_PUBLIC)
1952:     SPL_ME(Array, uksort, arginfo_array_uksort, ZEND_ACC_PUBLIC)
1953:     SPL_ME(Array, natsort, arginfo_array_void, ZEND_ACC_PUBLIC)
1954:     SPL_ME(Array, natcasesort, arginfo_array_void, ZEND_ACC_PUBLIC)
1955:     SPL_ME(Array, unserialize, arginfo_array_unserialize, ZEND_ACC_PUBLIC)
1956:     SPL_ME(Array, serialize, arginfo_array_void, ZEND_ACC_PUBLIC)
1957:     /* ArrayObject specific */
1958:     SPL_ME(Array, getIterator, arginfo_array_void, ZEND_ACC_PUBLIC)
1959:     SPL_ME(Array, exchangeArray, arginfo_array_exchangeArray, ZEND_ACC_PUBLIC)
1960:     SPL_ME(Array, setIteratorClass, arginfo_array_getIteratorClass, ZEND_ACC_PUBLIC)
1961:     SPL_ME(Array, getIteratorClass, arginfo_array_void, ZEND_ACC_PUBLIC)
1962:     PHP_FE_END
1963: };
1964:
1965: static const zend_function_entry spl_funcs_arrayIterator[] = {
1966:     SPL_ME(ArrayIterator, __construct, arginfo_array_iterator___construct, ZEND_ACC_PUBLIC)
1967:     SPL_ME(Array, offsetExists, arginfo_array_offsetGet, ZEND_ACC_PUBLIC)
1968:     SPL_ME(Array, offsetGet, arginfo_array_offsetGet, ZEND_ACC_PUBLIC)
1969:     SPL_ME(Array, offsetSet, arginfo_array_offsetSet, ZEND_ACC_PUBLIC)
1970:     SPL_ME(Array, offsetUnset, arginfo_array_offsetGet, ZEND_ACC_PUBLIC)
1971:     SPL_ME(Array, append, arginfo_array_append, ZEND_ACC_PUBLIC)
1972:     SPL_ME(Array, getArrayCopy, arginfo_array_void, ZEND_ACC_PUBLIC)
1973:     SPL_ME(Array, count, arginfo_array_void, ZEND_ACC_PUBLIC)
1974:     SPL_ME(Array, getFlags, arginfo_array_void, ZEND_ACC_PUBLIC)
1975:     SPL_ME(Array, setFlags, arginfo_array_setFlags, ZEND_ACC_PUBLIC)
1976:     SPL_ME(Array, asort, arginfo_array_void, ZEND_ACC_PUBLIC)
1977:     SPL_ME(Array, usort, arginfo_array_void, ZEND_ACC_PUBLIC)
1978:     SPL_ME(Array, uasort, arginfo_array_uksort, ZEND_ACC_PUBLIC)
1979:     SPL_ME(Array, uksort, arginfo_array_uksort, ZEND_ACC_PUBLIC)
1980:     SPL_ME(Array, natsort, arginfo_array_void, ZEND_ACC_PUBLIC)
1981:     SPL_ME(Array, natcasesort, arginfo_array_void, ZEND_ACC_PUBLIC)
1982:     SPL_ME(Array, unserialize, arginfo_array_unserialize, ZEND_ACC_PUBLIC)
1983:     SPL_ME(Array, serialize, arginfo_array_void, ZEND_ACC_PUBLIC)
1984:     /* ArrayIterator specific */
1985:     SPL_ME(Array, rewind, arginfo_array_void, ZEND_ACC_PUBLIC)
1986:     SPL_ME(Array, current, arginfo_array_void, ZEND_ACC_PUBLIC)
1987:     SPL_ME(Array, key, arginfo_array_void, ZEND_ACC_PUBLIC)
1988:     SPL_ME(Array, next, arginfo_array_void, ZEND_ACC_PUBLIC)
1989:     SPL_ME(Array, valid, arginfo_array_void, ZEND_ACC_PUBLIC)
1990:     SPL_ME(Array, seek, arginfo_array_seek, ZEND_ACC_PUBLIC)
1991:     PHP_FE_END
1992: };
1993:
1994: static const zend_function_entry spl_funcs_recursiveArrayIterator[] = {
1995:     SPL_ME(Array, hasChildren, arginfo_array_void, ZEND_ACC_PUBLIC)
1996:     SPL_ME(Array, getChildren, arginfo_array_void, ZEND_ACC_PUBLIC)
1997:     PHP_FE_END
1998: };
1999: /* }}} */
2000:
2001: /* {{{ PHP_MINIT_FUNCTION(spl_array) */
2002: PHP_MINIT_FUNCTION(spl_array)
2003: {
2004:     REGISTER_SPL_STD_CLASS_EX(ArrayObject, spl_array_object_new, spl_funcs_arrayObject);
2005:     REGISTER_SPL_IMPLEMENTATIONS(ArrayObject, Aggregate);
2006:     REGISTER_SPL_IMPLEMENTATIONS(ArrayObject, ArrayAccess);
2007:     REGISTER_SPL_IMPLEMENTATIONS(ArrayObject, Serializable);
2008:     REGISTER_SPL_IMPLEMENTATIONS(ArrayObject, Countable);
2009:     memcpy(spl_handler_arrayObject, zend_get_std_object_handlers(), sizeof(zend_object_handlers));
2010:
2011:     spl_handler_arrayObject->offset = XtOffsetOf(spl_array_object, std);
2012:
2013:     spl_handler_arrayObject->clone_obj = spl_array_object_clone;
2014:     spl_handler_arrayObject->read_dimension = spl_array_read_dimension;
2015:     spl_handler_arrayObject->write_dimension = spl_array_write_dimension;
2016:     spl_handler_arrayObject->unset_dimension = spl_array_unset_dimension;
2017:     spl_handler_arrayObject->has_dimension = spl_array_has_dimension;
2018:     spl_handler_arrayObject->count_elements = spl_array_object_count_elements;
2019:
2020:     spl_handler_arrayObject->get_properties = spl_array_get_properties;
2021:     spl_handler_arrayObject->get_debug_info = spl_array_get_debug_info;
2022:     spl_handler_arrayObject->get_gc = spl_array_get_gc;
2023:     spl_handler_arrayObject->read_property = spl_array_read_property;
2024:     spl_handler_arrayObject->write_property = spl_array_write_property;
2025:     spl_handler_arrayObject->get_property_ptr_ptr = spl_array_get_property_ptr_ptr;
2026:     spl_handler_arrayObject->has_property = spl_array_has_property;
2027:     spl_handler_arrayObject->unset_property = spl_array_unset_property;
2028:
2029:     spl_handler_arrayObject->compare_objects = spl_array_compare_objects;
2030:     spl_handler_arrayObject->dtor_obj = zend_objects_destroy_obj;
2031:     spl_handler_arrayObject->free_obj = spl_array_object_free_storage;
2032:
2033:     REGISTER_SPL_STD_CLASS_EX(ArrayIterator, spl_array_iterator_new, spl_funcs_arrayIterator);
2034:     REGISTER_SPL_IMPLEMENTATIONS(ArrayIterator, Iterator);
2035:     REGISTER_SPL_IMPLEMENTATIONS(ArrayIterator, ArrayAccess);
2036:     REGISTER_SPL_IMPLEMENTATIONS(ArrayIterator, SeekableIterator);
2037:     REGISTER_SPL_IMPLEMENTATIONS(ArrayIterator, Serializable);
2038:     REGISTER_SPL_IMPLEMENTATIONS(ArrayIterator, Countable);
2039:     memcpy(spl_handler_arrayIterator, spl_handler_arrayObject, sizeof(zend_object_handlers));
2040:     spl_array_iterator->get_iterator = spl_array_get_iterator;
2041:
2042:     REGISTER_SPL_CLASS_CONST_LONG(ArrayObject, "STD_PROP_LIST", SPL_ARRAY_STD_PROP_LIST);
2043:     REGISTER_SPL_CLASS_CONST_LONG(ArrayObject, "ARRAY_AS_PROPS", SPL_ARRAY_ARRAY_AS_PROPS);
2044:
2045:     REGISTER_SPL_CLASS_CONST_LONG(ArrayIterator, "STD_PROP_LIST", SPL_ARRAY_STD_PROP_LIST);
2046:     REGISTER_SPL_CLASS_CONST_LONG(ArrayIterator, "ARRAY_AS_PROPS", SPL_ARRAY_ARRAY_AS_PROPS);
2047:
2048:     REGISTER_SPL_SUB_CLASS_EX(RecursiveArrayIterator, ArrayIterator, spl_array_iterator_new, spl_funcs_recursiveArrayIterator);
2049:     REGISTER_SPL_IMPLEMENTATIONS(RecursiveArrayIterator, RecursiveIterator);
2050:     spl_recursiveArrayIterator->get_iterator = spl_array_get_iterator;
2051:
2052:     REGISTER_SPL_CLASS_CONST_LONG(RecursiveArrayIterator, "CHILD_ARRAYS_ONLY", SPL_ARRAY_CHILD_ARRAYS_ONLY);
2053:
2054:     return SUCCESS;
2055: }
2056: /* }}} */
2057:
2058: /*
2059:  * Local Variables:
2060:  * tab-width: 4
2061:  * c-basic-offset: 4
2062:  * End:
2063:  * vim600: fdm=marker
2064:  * vim: noet sw=4 ts=4
2065:  */

```

```
1: /*
2:  *-----*
3:  * | PHP Version 7 |
4:  *-----*
5:  * | Copyright (c) 1997-2018 The PHP Group |
6:  *-----*
7:  * | This source file is subject to version 3.01 of the PHP license, |
8:  * | that is bundled with this package in the file LICENSE, and is |
9:  * | available through the world-wide-web at the following url: |
10:  * | http://www.php.net/license/3.01.txt |
11:  * | If you did not receive a copy of the PHP license and are unable to |
12:  * | obtain it through the world-wide-web, please send a note to |
13:  * | license@php.net so we can mail you a copy immediately. |
14:  *-----*
15:  * | Authors: Etienne Kneuss <colder@php.net> |
16:  *-----*
17:  */
18:
19: /* $Id$ */
20:
21: #ifndef SPL_HEAP_H
22: #define SPL_HEAP_H
23:
24: #include "php.h"
25: #include "php_spl.h"
26:
27: extern PHPAPI zend_class_entry *spl_ce_SplHeap;
28: extern PHPAPI zend_class_entry *spl_ce_SplMinHeap;
29: extern PHPAPI zend_class_entry *spl_ce_SplMaxHeap;
30:
31: extern PHPAPI zend_class_entry *spl_ce_SplPriorityQueue;
32:
33: PHP_MINIT_FUNCTION(spl_heap);
34:
35: #endif /* SPL_HEAP_H */
36:
37: /*
38:  * Local Variables:
39:  * c-basic-offset: 4
40:  * tab-width: 4
41:  * End:
42:  * vim600: fdm=marker
43:  * vim: noet sw=4 ts=4
44:  */
```

```
1: /*
2:  *-----*
3:  * PHP Version 7
4:  *-----*
5:  * Copyright (c) 1997-2018 The PHP Group
6:  *-----*
7:  * This source file is subject to version 3.01 of the PHP license,
8:  * that is bundled with this package in the file LICENSE, and is
9:  * available through the world-wide-web at the following url:
10:  * http://www.php.net/license/3.01.txt
11:  * If you did not receive a copy of the PHP license and are unable to
12:  * obtain it through the world-wide-web, please send a note to
13:  * license@php.net so we can mail you a copy immediately.
14:  *-----*
15:  * Authors: Marcus Boerger <helly@php.net>
16:  *-----*
17:  */
18:
19: /* $Id$ */
20:
21: #ifndef SPL_ENGINE_H
22: #define SPL_ENGINE_H
23:
24: #include "php.h"
25: #include "php_spl.h"
26: #include "zend_interfaces.h"
27:
28: PHPAPI void spl_instantiate(zend_class_entry *pce, zval *object);
29:
30: PHPAPI zend_long spl_offset_convert_to_long(zval *offset);
31:
32: /* {{{ spl_instantiate_arg_ex1 */
33: static inline int spl_instantiate_arg_ex1(zend_class_entry *pce, zval *retval, zval *arg1)
34: {
35:     zend_function *func = pce->constructor;
36:     spl_instantiate(pce, retval);
37:
38:     zend_call_method(retval, pce, &func, ZEND_VAL(func->common.function_name), ZEND_LEN(func->common.function_name), NULL, 1, arg1, NULL);
39:     return 0;
40: }
41: /* }}} */
42:
43: /* {{{ spl_instantiate_arg_ex2 */
44: static inline int spl_instantiate_arg_ex2(zend_class_entry *pce, zval *retval, zval *arg1, zval *arg2)
45: {
46:     zend_function *func = pce->constructor;
47:     spl_instantiate(pce, retval);
48:
49:     zend_call_method(retval, pce, &func, ZEND_VAL(func->common.function_name), ZEND_LEN(func->common.function_name), NULL, 2, arg1, arg2);
50:     return 0;
51: }
52: /* }}} */
53:
54: /* {{{ spl_instantiate_arg_n */
55: static inline void spl_instantiate_arg_n(zend_class_entry *pce, zval *retval, int argc, zval *argv)
56: {
57:     zend_function *func = pce->constructor;
58:     zend_fcall_info fci;
59:     zend_fcall_info_cache fcc;
60:     zval dummy;
61:
62:     spl_instantiate(pce, retval);
63:
64:     fci.size = sizeof(zend_fcall_info);
65:     ZVAL_STR(&fci.function_name, func->common.function_name);
66:     fci.object = Z_OBJ_P(retval);
67:     fci.retval = &dummy;
68:     fci.param_count = argc;
69:     fci.params = argv;
70:     fci.no_separation = 1;
71:
72:     fcc.function_handler = func;
73:     fcc.calling_scope = zend_get_executed_scope();
74:     fcc.called_scope = pce;
75:     fcc.object = Z_OBJ_P(retval);
76:
77:     zend_call_function(&fci, &fcc);
78: }
79: /* }}} */
80:
81: #endif /* SPL_ENGINE_H */
82:
83: /*
84:  * Local Variables:
85:  * n-basis-offset: 4
86:  * tab-width: 4
87:  * End:
88:  * vim600: fdm=marker
89:  * vim: noet sw=4 ts=4
90:  */
```





```

376:     } else {
377:         zend_clear_exception();
378:     }
379: }
380: }
381: goto next_step;
382: }
383: /* no more elements */
384: if (object->level > 0) {
385:     if (object->endChildren) {
386:         zend_call_method_with_0_params(&this, object->ce, object->zendChildren, "endchildren", NULL);
387:     }
388:     if (EG(exception)) {
389:         if (!object->flags & RIT_CATCH_GET_CHILD) {
390:             return;
391:         } else {
392:             zend_clear_exception();
393:         }
394:     }
395:     if (object->level > 0) {
396:         zval garbage;
397:         ZVAL_COPY_VALUE(&garbage, sub_iter->iterators[object->level].sub_iter);
398:         ZVAL_UNDEF(&object->iterators[object->level].sub_iter);
399:         zval_ptr_dtor(&garbage);
400:         zend_iterator_dtor(iterator);
401:         object->level--;
402:     }
403: } else {
404:     return; /* done completely */
405: }
406: }
407: }
408:
409: static void spl_recursive_it_rewind_ex(spl_recursive_it_object *object, zval *rthis)
410: {
411:     zend_object_iterator *sub_iter;
412:
413:     SPL_FETCH_SUB_ITERATOR(sub_iter, object);
414:
415:     while (object->level) {
416:         sub_iter = object->iterators[object->level].iterator;
417:         zend_iterator_dtor(sub_iter);
418:         zval_ptr_dtor(&object->iterators[object->level-1].sub_iter);
419:         if (EG(exception) && (object->endChildren || object->endChildren->common.scope != spl_ce_RecursiveIteratorIterator)) {
420:             zend_call_method_with_0_params(&this, object->ce, object->zendChildren, "endchildren", NULL);
421:         }
422:     }
423:     object->iterators = erealloc(object->iterators, sizeof(spl_sub_iterator));
424:     object->iterators[0].state = RS_START;
425:     sub_iter = object->iterators[0].iterator;
426:     if (sub_iter->funcs->rewind) {
427:         sub_iter->funcs->rewind(sub_iter);
428:     }
429:     if (EG(exception) && object->beginIteration == 1) {
430:         zend_call_method_with_0_params(&this, object->ce, object->beginIteration, "beginIteration", NULL);
431:     }
432:     object->in_iteration = 1;
433:     spl_recursive_it_move_forward_ex(object, rthis);
434: }
435:
436: static void spl_recursive_it_move_forward(spl_recursive_it_iterator *iter)
437: {
438:     spl_recursive_it_move_forward_ex(&iter->data, &iter->data);
439: }
440:
441: static void spl_recursive_it_rewind(spl_recursive_it_iterator *iter)
442: {
443:     spl_recursive_it_rewind_ex(&iter->data, &iter->data);
444: }
445:
446: static zend_object_iterator *spl_recursive_it_get_iterator(spl_class_entry *ce, zval *object, int by_ref)
447: {
448:     spl_recursive_it_iterator *iterator;
449:     spl_recursive_it_object *object;
450:
451:     if (by_ref) {
452:         zend_throw_exception(spl_ce_RuntimeException, "An iterator cannot be used with foreach by reference", 0);
453:         return NULL;
454:     }
455:     iterator = emalloc(sizeof(spl_recursive_it_iterator));
456:     object = &iter->data;
457:     if (object->iterators == NULL) {
458:         zend_error(E_ERROR, "The object to be iterated is in an invalid state: "
459:             "the parent constructor has not been called");
460:     }
461:
462:     zend_iterator_init(&zend_object_iterator * iterator);
463:
464:     ZVAL_COPY(&iterator->intern.data, &object);
465:     iterator->intern.funcs = ce->iterator_funcs.funcs;
466:     return (&zend_object_iterator * iterator);
467: }
468:
469: static const zend_object_iterator_funcs spl_recursive_it_iterator_funcs = {
470:     spl_recursive_it_dtor,
471:     spl_recursive_it_valid,
472:     spl_recursive_it_get_current_data,
473:     spl_recursive_it_get_current_key,
474:     spl_recursive_it_move_forward,
475:     spl_recursive_it_rewind,
476:     NULL
477: };
478:
479: static void spl_recursive_it_it_construct(INTERNAL_FUNCTION_PARAMETERS, zend_class_entry *ce_base, zend_class_entry *ce_inner, recursive_it_type rit_type)
480: {
481:     zval *object = getThis();
482:     spl_recursive_it_object *intern;
483:     zval *iterator;
484:     zend_class_entry *ce_iterator;
485:     zend_long mode, flags;
486:     zend_error_handling error_handling;
487:     zval caching_it, aggregate_retval;
488:
489:     zend_replace_error_handling(EH_THROW, spl_ce_InvalidArgumentException, &error_handling);
490:
491:     switch (rit_type) {
492:         case RIT_RecursiveForAggregate: {
493:             zval caching_it_flags, *user_caching_it_flags = NULL;
494:             mode = RIT_HELP_FIRST;
495:             flags = RIT_BYPASS_RIT;
496:
497:             if (zend_parse_parameters_ex(ZEND_PARSE_PARAMS_QUIET, ZEND_NUM_ARGS(), "o|l", iterator, &mode, &flags, &user_caching_it_flags, &mode) == SUCCESS) {
498:                 if (instanceof_function(&OBJECT_P(iterator), &ce_base, &ce_inner)) {
499:                     zend_call_method_with_0_params(iterator, &OBJECT_P(iterator), &OBJECT_P(iterator)->iterator_funcs.if_new_iterator, "getIterator", &aggregate_retval);
500:
501:                     iterator = &aggregate_retval;
502:                 } else {
503:                     Z_ADDREF_P(iterator);
504:                 }
505:
506:                 if (user_caching_it_flags) {
507:                     ZVAL_COPY(&caching_it_flags, user_caching_it_flags);
508:                 } else {
509:                     ZVAL_LONG(&caching_it_flags, CTT_CATCH_GET_CHILD);
510:                 }
511:                 spl_instantiate_arg_ex2(spl_ce_RecursiveCachingIterator, &caching_it, iterator, &caching_it_flags);
512:                 zval_ptr_dtor(&caching_it_flags);
513:                 zval_ptr_dtor(iterator);
514:                 iterator = &caching_it;
515:             } else {
516:                 iterator = NULL;
517:             }
518:             break;
519:         }
520:         case RIT_RecursiveIteratorIterator: {
521:             default: {
522:                 mode = RIT_LEAVES_ONLY;
523:                 flags = 0;
524:
525:                 if (zend_parse_parameters_ex(ZEND_PARSE_PARAMS_QUIET, ZEND_NUM_ARGS(), "o|l", iterator, &mode, &flags) == SUCCESS) {
526:                     if (instanceof_function(&OBJECT_P(iterator), &ce_base, &ce_inner)) {
527:                         zend_call_method_with_0_params(iterator, &OBJECT_P(iterator), &OBJECT_P(iterator)->iterator_funcs.if_new_iterator, "getIterator", &aggregate_retval);
528:
529:                         iterator = &aggregate_retval;
530:                     } else {
531:                         Z_ADDREF_P(iterator);
532:                     }
533:                 } else {
534:                     iterator = NULL;
535:                 }
536:                 break;
537:             }
538:         }
539:         if (iterator) {
540:             zval_ptr_dtor(iterator);
541:         }
542:         zend_throw_exception(spl_ce_InvalidArgumentException, "An instance of RecursiveIterator or IteratorAggregate creating it is required", 0);
543:         zend_restore_error_handling(&error_handling);
544:         return;
545:     }
546:
547:     intern = &SPL_RECURSIVE_IT_P(object);
548:     intern->iterators = emalloc(sizeof(spl_sub_iterator));
549:     intern->level = 0;
550:     intern->mode = mode;
551:     intern->flags = (int) flags;
552:     intern->max_depth = -1;
553:     intern->in_iteration = 0;
554:     intern->ce = &OBJECT_P(object);
555:
556:     intern->beginIteration = zend_hash_str_find_ptr(intern->ce->function_table, "beginIteration", sizeof("beginIteration") - 1);
557:     if (intern->beginIteration->common.scope == ce_base) {
558:         intern->beginIteration = NULL;
559:     }
560:     intern->endIteration = zend_hash_str_find_ptr(intern->ce->function_table, "endIteration", sizeof("endIteration") - 1);
561:     if (intern->endIteration->common.scope == ce_base) {
562:         intern->endIteration = NULL;
563:     }
564:     intern->callHasChildren = zend_hash_str_find_ptr(intern->ce->function_table, "callHasChildren", sizeof("callHasChildren") - 1);
565:     if (intern->callHasChildren->common.scope == ce_base) {
566:         intern->callHasChildren = NULL;
567:     }
568:     intern->callGetChildren = zend_hash_str_find_ptr(intern->ce->function_table, "callGetChildren", sizeof("callGetChildren") - 1);
569:     if (intern->callGetChildren->common.scope == ce_base) {
570:         intern->callGetChildren = NULL;
571:     }
572:     intern->beginChildren = zend_hash_str_find_ptr(intern->ce->function_table, "beginChildren", sizeof("beginChildren") - 1);
573:     if (intern->beginChildren->common.scope == ce_base) {
574:         intern->beginChildren = NULL;
575:     }
576:     intern->endChildren = zend_hash_str_find_ptr(intern->ce->function_table, "endChildren", sizeof("endChildren") - 1);
577:     if (intern->endChildren->common.scope == ce_base) {
578:         intern->endChildren = NULL;
579:     }
580:     intern->nextElement = zend_hash_str_find_ptr(intern->ce->function_table, "nextElement", sizeof("nextElement") - 1);
581:     if (intern->nextElement->common.scope == ce_base) {
582:         intern->nextElement = NULL;
583:     }
584:
585:     ce_iterator = &OBJECT_P(iterator); /* respect inheritance, don't use spl_ce_RecursiveIterator */
586:     intern->iterators[0].iterator = ce_iterator->get_iterator(ce_iterator, iterator, 0);
587:     ZVAL_COPY_VALUE(intern->iterators[0].sub_iter, iterator);
588:     intern->iterators[0].ce = ce_iterator;
589:     intern->iterators[0].state = RS_START;
590:
591:     zend_restore_error_handling(&error_handling);
592:
593:     if (EG(exception)) {
594:         zend_object_iterator *sub_iter;
595:
596:         while (intern->level > 0) {
597:             sub_iter = intern->iterators[intern->level].iterator;
598:             zend_iterator_dtor(sub_iter);
599:             zval_ptr_dtor(&intern->iterators[intern->level-1].sub_iter);
600:         }
601:         ifree(intern->iterators);
602:         intern->iterators = NULL;
603:     }
604: }
605:
606: /* {{{ proto void RecursiveIteratorIterator::__construct(RecursiveIteratorIteratorAggregate $it, $int mode = RIT_LEAVES_ONLY, $int flags = 0) throw $InvalidArgumentException
607: Creates a RecursiveIteratorIterator from a RecursiveIterator. */
608: SPL_METHOD(RecursiveIteratorIterator, __construct)
609: {
610:     spl_recursive_it_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ce_RecursiveIteratorIterator, &zend_object_iterator, RIT_RecursiveIteratorIterator);
611: } /* }}} */
612:
613: /* {{{ proto void RecursiveIteratorIterator::rewind()
614: Rewind the iterator to the first element of the top level inner iterator. */
615: SPL_METHOD(RecursiveIteratorIterator, rewind)
616: {
617:     spl_recursive_it_object *object = &SPL_RECURSIVE_IT_P(getThis());
618:
619:     if (zend_parse_parameters_none() == FAILURE) {
620:         return;
621:     }
622:
623:     spl_recursive_it_rewind_ex(object, getThis());
624: } /* }}} */
625:
626: /* {{{ proto bool RecursiveIteratorIterator::valid()
627: Check whether the current position is valid */
628: SPL_METHOD(RecursiveIteratorIterator, valid)
629: {
630:     spl_recursive_it_object *object = &SPL_RECURSIVE_IT_P(getThis());
631:
632:     if (zend_parse_parameters_none() == FAILURE) {
633:         return;
634:     }
635:
636:     RETURN_BOOL(spl_recursive_it_valid_ex(object, getThis()) == SUCCESS);
637: } /* }}} */
638:
639: /* {{{ proto mixed RecursiveIteratorIterator::key()
640: Access the current key */
641: SPL_METHOD(RecursiveIteratorIterator, key)
642: {
643:     spl_recursive_it_object *object = &SPL_RECURSIVE_IT_P(getThis());
644:     zend_object_iterator *iterator;
645:
646:     if (zend_parse_parameters_none() == FAILURE) {
647:         return;
648:     }
649:
650:     SPL_FETCH_SUB_ITERATOR(iterator, object);
651:
652:     if (iterator->funcs->get_current_key) {
653:         iterator->funcs->get_current_key(iterator, &return_value);
654:     } else {
655:         RETURN_NULL();
656:     }
657: } /* }}} */
658:
659: /* {{{ proto mixed RecursiveIteratorIterator::current()
660: Access the current element value */
661: SPL_METHOD(RecursiveIteratorIterator, current)
662: {
663:     spl_recursive_it_object *object = &SPL_RECURSIVE_IT_P(getThis());
664:     zend_object_iterator *iterator;
665:     zval *data;
666:
667:     if (zend_parse_parameters_none() == FAILURE) {
668:         return;
669:     }
670:
671:     SPL_FETCH_SUB_ITERATOR(iterator, object);
672:
673:     data = iterator->funcs->get_current_data(iterator);
674:     if (data) {
675:         ZVAL_DEREF(data);
676:         ZVAL_COPY(&return_value, data);
677:     }
678: } /* }}} */
679:
680: /* {{{ proto void RecursiveIteratorIterator::next()
681: Move forward to the next element */
682: SPL_METHOD(RecursiveIteratorIterator, next)
683: {
684:     spl_recursive_it_object *object = &SPL_RECURSIVE_IT_P(getThis());
685:
686:     if (zend_parse_parameters_none() == FAILURE) {
687:         return;
688:     }
689:
690:     spl_recursive_it_move_forward_ex(object, getThis());
691: } /* }}} */
692:
693: /* {{{ proto int RecursiveIteratorIterator::getDepth()
694: Get the current depth of the recursive iteration */
695: SPL_METHOD(RecursiveIteratorIterator, getDepth)
696: {
697:     spl_recursive_it_object *object = &SPL_RECURSIVE_IT_P(getThis());
698:
699:     if (zend_parse_parameters_none() == FAILURE) {
700:         return;
701:     }
702:
703:     RETURN_LONG(object->level);
704: } /* }}} */
705:
706: /* {{{ proto RecursiveIterator RecursiveIteratorIterator::getSubIterator($int level)
707: The current active sub iterator or the iterator at specified level */
708: SPL_METHOD(RecursiveIteratorIterator, getSubIterator)
709: {
710:     spl_recursive_it_object *object = &SPL_RECURSIVE_IT_P(getThis());
711:     zend_long level = object->level;
712:     zval *value;
713:
714:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "[i]", &level) == FAILURE) {
715:         return;
716:     }
717:     if (level < 0 || level > object->level) {
718:         RETURN_NULL();
719:     }
720:
721:     if (object->iterators) {
722:         zend_throw_exception(spl_ce_LogicException, 0,
723:             "The object is in an invalid state as the parent constructor was not called");
724:         return;
725:     }
726:
727:     value = object->iterators[level].sub_iter;
728:     ZVAL_DEREF(value);
729:     ZVAL_COPY(&return_value, value);
730: } /* }}} */
731:
732: /* {{{ proto RecursiveIterator RecursiveIteratorIterator::getInnerIterator()
733: The current active sub iterator */
734: SPL_METHOD(RecursiveIteratorIterator, getInnerIterator)
735: {
736:     spl_recursive_it_object *object = &SPL_RECURSIVE_IT_P(getThis());
737:     zval *sub_iter;
738:
739:     if (zend_parse_parameters_none() == FAILURE) {
740:         return;
741:     }
742:
743:     SPL_FETCH_SUB_ELEMENT_ADDR(sub_iter, object, sub_iter);
744:
745:     ZVAL_DEREF(sub_iter);
746:     ZVAL_COPY(&return_value, sub_iter);
747: } /* }}} */

```

```

748: /* {{{ proto RecursiveIterator RecursiveIteratorIterator::beginIteration()
749: Called when iteration begins (after first rewind() call) */
750: SPL_METHOD(RecursiveIteratorIterator, beginIteration)
751: {
752:     IF (zend_parse_parameters_none() == FAILURE) {
753:         return;
754:     }
755:     /* nothing to do */
756:     /* }}} */
757: }
758:
759: /* {{{ proto RecursiveIterator RecursiveIteratorIterator::endIteration()
760: Called when iteration ends (when valid() first returns false) */
761: SPL_METHOD(RecursiveIteratorIterator, endIteration)
762: {
763:     IF (zend_parse_parameters_none() == FAILURE) {
764:         return;
765:     }
766:     /* nothing to do */
767:     /* }}} */
768: }
769: /* {{{ proto bool RecursiveIteratorIterator::callHasChildren()
770: Called for each element to test whether it has children */
771: SPL_METHOD(RecursiveIteratorIterator, callHasChildren)
772: {
773:     spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(getThis());
774:     zend_class_entry *ce;
775:     zval *obj;
776:
777:     IF (zend_parse_parameters_none() == FAILURE) {
778:         return;
779:     }
780:
781:     IF (obj->iterators) {
782:         RETURN_NULL();
783:     }
784:
785:     SPL_FETCH_SUB_ELEMENT(ce, object, ce);
786:
787:     obj = obj->iterators[object->level].obj;
788:     IF (Z_TYPE_P(obj) == IS_UNDEF) {
789:         RETURN_FALSE;
790:     } else {
791:         zend_call_method_with_0_params(obj, ce, NULL, "hasChildren", return_value);
792:         IF (Z_TYPE_P(return_value) == IS_UNDEF) {
793:             RETURN_FALSE;
794:         }
795:     }
796:     /* }}} */
797: }
798: /* {{{ proto RecursiveIterator RecursiveIteratorIterator::callGetChildren()
799: Return children of current element */
800: SPL_METHOD(RecursiveIteratorIterator, callGetChildren)
801: {
802:     spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(getThis());
803:     zend_class_entry *ce;
804:     zval *obj;
805:
806:     IF (zend_parse_parameters_none() == FAILURE) {
807:         return;
808:     }
809:
810:     SPL_FETCH_SUB_ELEMENT(ce, object, ce);
811:
812:     obj = obj->iterators[object->level].obj;
813:     IF (Z_TYPE_P(obj) == IS_UNDEF) {
814:         return;
815:     } else {
816:         zend_call_method_with_0_params(obj, ce, NULL, "getChildren", return_value);
817:         IF (Z_TYPE_P(return_value) == IS_UNDEF) {
818:             RETURN_NULL();
819:         }
820:     }
821:     /* }}} */
822: }
823: /* {{{ proto void RecursiveIteratorIterator::beginChildren()
824: Called when recursing one level down */
825: SPL_METHOD(RecursiveIteratorIterator, beginChildren)
826: {
827:     IF (zend_parse_parameters_none() == FAILURE) {
828:         return;
829:     }
830:     /* nothing to do */
831:     /* }}} */
832: }
833: /* {{{ proto void RecursiveIteratorIterator::endChildren()
834: Called when end recursing one level */
835: SPL_METHOD(RecursiveIteratorIterator, endChildren)
836: {
837:     IF (zend_parse_parameters_none() == FAILURE) {
838:         return;
839:     }
840:     /* nothing to do */
841:     /* }}} */
842: }
833: /* {{{ proto void RecursiveIteratorIterator::nextElement()
834: Called when the next element is available */
835: SPL_METHOD(RecursiveIteratorIterator, nextElement)
836: {
837:     IF (zend_parse_parameters_none() == FAILURE) {
838:         return;
839:     }
840:     /* nothing to do */
841:     /* }}} */
842: }
843: /* {{{ proto void RecursiveIteratorIterator::setMaxDepth()
844: Set the maximum allowed depth (or any depth if max_depth = -1) */
845: SPL_METHOD(RecursiveIteratorIterator, setMaxDepth)
846: {
847:     IF (zend_parse_parameters_none() == FAILURE) {
848:         return;
849:     }
850:     /* nothing to do */
851:     /* }}} */
852: }
853: /* {{{ proto void RecursiveIteratorIterator::setMaxDepth(int $max_depth = -1)
854: Set the maximum allowed depth (or any depth if max_depth = -1) */
855: SPL_METHOD(RecursiveIteratorIterator, setMaxDepth)
856: {
857:     spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(getThis());
858:     zend_long max_depth = -1;
859:
860:     IF (zend_parse_parameters(ZEND_NUM_ARGS(), "i", &max_depth) == FAILURE) {
861:         return;
862:     }
863:     IF (max_depth < -1) {
864:         zend_throw_exception(spl_ce_OutOfRangeException, "Parameter max_depth must be >= -1", 0);
865:         return;
866:     } else IF (max_depth > INT_MAX) {
867:         max_depth = INT_MAX;
868:     }
869:
870:     object->max_depth = (int)max_depth;
871:     /* }}} */
872: }
873: /* {{{ proto int RecursiveIteratorIterator::getMaxDepth()
874: Return the maximum accepted depth or false if any depth is allowed */
875: SPL_METHOD(RecursiveIteratorIterator, getMaxDepth)
876: {
877:     spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(getThis());
878:
879:     IF (zend_parse_parameters_none() == FAILURE) {
880:         return;
881:     }
882:
883:     IF (object->max_depth == -1) {
884:         RETURN_FALSE;
885:     } else {
886:         RETURN_LONG(object->max_depth);
887:     }
888:     /* }}} */
889: }
890:
891: static union zend_function *spl_recursive_it_get_method(zend_object **obj, zend_string *method, const zval *key)
892: {
893:     union_zend_function *function_handler;
894:     spl_recursive_it_object *object = spl_recursive_it_from_obj(*obj);
895:     zend_long level = object->level;
896:     zval *obj;
897:
898:     IF (obj->iterators) {
899:         php_error_docref(NULL, E_WARNING, "The instance wasn't initialized properly", ZSTR_VAL(*obj), "ce->name");
900:     }
901:     obj = obj->iterators[level].obj;
902:     function_handler = std_object_handlers.get_method(obj, method, key);
903:     IF (function_handler) {
904:         zend_hash_find_ptr(&Z_OBJ_HT_P(obj)->function_table, method) == NULL;
905:         IF (Z_OBJ_HT_P(obj)->get_method() {
906:             *obj->get_method() = Z_OBJ_P(obj);
907:             function_handler = (*obj->handlers)->get_method(obj, method, key);
908:         } else {
909:             *obj->get_method() = Z_OBJ_P(obj);
910:         }
911:     }
912:     return function_handler;
913: }
914:
915: /* {{{ spl_recursive_iterator_iterator_ctor */
916: static void spl_recursive_iterator_iterator_ctor(zend_object **obj)
917: {
918:     spl_recursive_it_object *object = spl_recursive_it_from_obj(*obj);
919:     zend_object_iterator *sub_iter;
920:
921:     /* call standard ctor */
922:     zend_objects_destroy_object(*obj);
923:
924:     IF (obj->iterators) {
925:         while (object->level >= 0) {
926:             sub_iter = obj->iterators[object->level].iterator;
927:             zend_iterator_ctor(sub_iter);
928:             zval_ptr_dtor(&obj->iterators[object->level--].obj);
929:         }
930:     }
931:     efree(obj->iterators);
932:     obj->iterators = NULL;
933: }
934:
935: /* }}} */

```

```

1124: smart_str_free(object->prefix[part]);
1125:
1126: smart_str_append(object->prefix[part], prefix, prefix_len);
1127: /* '}' */
1128:
1129: /* {{{ proto string RecursiveTreeIterator::getPrefix()
1130:  Returns the string to place in front of current element */
1131: SPL_METHOD (RecursiveTreeIterator, getPrefix)
1132: {
1133:     spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(getThis());
1134:
1135:     if (zend_parse_parameters_none() == FAILURE) {
1136:         return;
1137:     }
1138:
1139:     if(!object->itersators) {
1140:         zend_throw_exception_ex(spl_ce_LogicException, 0,
1141:             "The object is in an invalid state as the parent constructor was not called");
1142:         return;
1143:     }
1144:
1145:     spl_recursive_tree_iterator_get_prefix(object, return_value);
1146:     /* '}' */
1147:
1148:     /* {{{ proto void RecursiveTreeIterator::setPostfix(string prefix)
1149:      Sets postfix as used in getPrefix() */
1150: SPL_METHOD (RecursiveTreeIterator, setPostfix)
1151: {
1152:     spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(getThis());
1153:     char* postfix;
1154:     size_t postfix_len;
1155:
1156:     if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "s", &postfix, &postfix_len) == FAILURE) {
1157:         return;
1158:     }
1159:
1160:     smart_str_free(object->postfix[0]);
1161:     smart_str_append(object->postfix[0], postfix, postfix_len);
1162:     /* '}' */
1163:
1164:     /* {{{ proto string RecursiveTreeIterator::getEntry()
1165:  Returns the string presentation built for current element */
1166: SPL_METHOD (RecursiveTreeIterator, getEntry)
1167: {
1168:     spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(getThis());
1169:
1170:     if (zend_parse_parameters_none() == FAILURE) {
1171:         return;
1172:     }
1173:
1174:     if(!object->itersators) {
1175:         zend_throw_exception_ex(spl_ce_LogicException, 0,
1176:             "The object is in an invalid state as the parent constructor was not called");
1177:         return;
1178:     }
1179:
1180:     spl_recursive_tree_iterator_get_entry(object, return_value);
1181:     /* '}' */
1182:
1183:     /* {{{ proto string RecursiveTreeIterator::getPostfix()
1184:  Returns the string to place after the current element */
1185: SPL_METHOD (RecursiveTreeIterator, getPostfix)
1186: {
1187:     spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(getThis());
1188:
1189:     if (zend_parse_parameters_none() == FAILURE) {
1190:         return;
1191:     }
1192:
1193:     if(!object->itersators) {
1194:         zend_throw_exception_ex(spl_ce_LogicException, 0,
1195:             "The object is in an invalid state as the parent constructor was not called");
1196:         return;
1197:     }
1198:
1199:     spl_recursive_tree_iterator_get_postfix(object, return_value);
1200:     /* '}' */
1201:
1202:     /* {{{ proto mixed RecursiveTreeIterator::current()
1203:  Returns the current element prefixed and postfixed */
1204: SPL_METHOD (RecursiveTreeIterator, current)
1205: {
1206:     spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(getThis());
1207:     zval prefix, entry, postfix;
1208:     char *ptr;
1209:     zend_string *str;
1210:
1211:     if (zend_parse_parameters_none() == FAILURE) {
1212:         return;
1213:     }
1214:
1215:     if(!object->itersators) {
1216:         zend_throw_exception_ex(spl_ce_LogicException, 0,
1217:             "The object is in an invalid state as the parent constructor was not called");
1218:         return;
1219:     }
1220:
1221:     if (object->flags & BIT1_BYPASS_CURRENT) {
1222:         zend_object_iterator *iterator = object->itersators[object->level].iterator;
1223:         zval *data;
1224:
1225:         SPL_FETCH_SUB_ITERATOR(iterator, object);
1226:         data = iterator->funcs->get_current_data(iterator);
1227:         if (data) {
1228:             ZVAL_DEREF(data);
1229:             ZVAL_COPY(return_value, data);
1230:             return;
1231:         } else {
1232:             RETURN_NULL();
1233:         }
1234:     }
1235:
1236:     ZVAL_NULL(&prefix);
1237:     ZVAL_NULL(&entry);
1238:     spl_recursive_tree_iterator_get_prefix(object, &prefix);
1239:     spl_recursive_tree_iterator_get_entry(object, &entry);
1240:     if (Z_TYPE(entry) != IS_STRING) {
1241:         zval_ptr_dtor(&prefix);
1242:         zval_ptr_dtor(&entry);
1243:         RETURN_NULL();
1244:     }
1245:     spl_recursive_tree_iterator_get_postfix(object, &postfix);
1246:
1247:     str = zend_string_alloc(Z_STRLEN(prefix) + Z_STRLEN(entry) + Z_STRLEN(postfix), 0);
1248:     ptr = ZSTR_VAL(str);
1249:
1250:     memcpy(ptr, Z_STRVAL(prefix), Z_STRLEN(prefix));
1251:     ptr += Z_STRLEN(prefix);
1252:     memcpy(ptr, Z_STRVAL(entry), Z_STRLEN(entry));
1253:     ptr += Z_STRLEN(entry);
1254:     memcpy(ptr, Z_STRVAL(postfix), Z_STRLEN(postfix));
1255:     ptr += Z_STRLEN(postfix);
1256:     *ptr = 0;
1257:
1258:     zval_ptr_dtor(&prefix);
1259:     zval_ptr_dtor(&entry);
1260:     zval_ptr_dtor(&postfix);
1261:
1262:     RETURN_NEW_STR(str);
1263:     /* '}' */
1264:
1265:     /* {{{ proto mixed RecursiveTreeIterator::key()
1266:  Returns the current key prefixed and postfixed */
1267: SPL_METHOD (RecursiveTreeIterator, key)
1268: {
1269:     spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(getThis());
1270:     zend_object_iterator *iterator;
1271:     zval prefix, key, postfix, key_copy;
1272:     char *ptr;
1273:     zend_string *str;
1274:
1275:     if (zend_parse_parameters_none() == FAILURE) {
1276:         return;
1277:     }
1278:
1279:     SPL_FETCH_SUB_ITERATOR(iterator, object);
1280:
1281:     if (iterator->funcs->get_current_key) {
1282:         iterator->funcs->get_current_key(iterator, &key);
1283:     } else {
1284:         ZVAL_NULL(&key);
1285:     }
1286:
1287:     if (object->flags & BIT1_BYPASS_KEY) {
1288:         RETVAL_ZVAL(&key, 1, 1);
1289:         return;
1290:     }
1291:
1292:     if (Z_TYPE(key) != IS_STRING) {
1293:         if (zend_make_printable_zval(&key, &key_copy)) {
1294:             key = key_copy;
1295:         }
1296:     }
1297:
1298:     spl_recursive_tree_iterator_get_prefix(object, &prefix);
1299:     spl_recursive_tree_iterator_get_postfix(object, &postfix);
1300:
1301:     str = zend_string_alloc(Z_STRLEN(prefix) + Z_STRLEN(key) + Z_STRLEN(postfix), 0);
1302:     ptr = ZSTR_VAL(str);
1303:
1304:     memcpy(ptr, Z_STRVAL(prefix), Z_STRLEN(prefix));
1305:     ptr += Z_STRLEN(prefix);
1306:     memcpy(ptr, Z_STRVAL(key), Z_STRLEN(key));
1307:     ptr += Z_STRLEN(key);
1308:     memcpy(ptr, Z_STRVAL(postfix), Z_STRLEN(postfix));
1309:     ptr += Z_STRLEN(postfix);
1310:     *ptr = 0;
1311:
1312:     zval_ptr_dtor(&prefix);
1313:     zval_ptr_dtor(&key);
1314:     zval_ptr_dtor(&postfix);
1315:
1316:     RETURN_NEW_STR(str);
1317:     /* '}' */
1318:
1319:     /* {{{ proto void RecursiveTreeIterator::rewind()
1320:  Rewind the iterator to the first element */
1321: SPL_METHOD (RecursiveTreeIterator, rewind)
1322: {
1323:     spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(getThis());
1324:
1325:     if (zend_parse_parameters_none() == FAILURE) {
1326:         return;
1327:     }
1328:
1329:     if(!object->itersators) {
1330:         zend_throw_exception_ex(spl_ce_LogicException, 0,
1331:             "The object is in an invalid state as the parent constructor was not called");
1332:         return;
1333:     }
1334:
1335:     spl_recursive_tree_iterator_rewind(object);
1336:     /* '}' */
1337:
1338:     /* {{{ proto void RecursiveTreeIterator::valid()
1339:  Checks if the current element is valid */
1340: SPL_METHOD (RecursiveTreeIterator, valid)
1341: {
1342:     spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(getThis());
1343:
1344:     if (zend_parse_parameters_none() == FAILURE) {
1345:         return;
1346:     }
1347:
1348:     if(!object->itersators) {
1349:         zend_throw_exception_ex(spl_ce_LogicException, 0,
1350:             "The object is in an invalid state as the parent constructor was not called");
1351:         return;
1352:     }
1353:
1354:     spl_recursive_tree_iterator_valid(object);
1355:     /* '}' */
1356:
1357:     /* {{{ proto void RecursiveTreeIterator::current()
1358:  Returns the current element */
1359: SPL_METHOD (RecursiveTreeIterator, current)
1360: {
1361:     spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(getThis());
1362:
1363:     if (zend_parse_parameters_none() == FAILURE) {
1364:         return;
1365:     }
1366:
1367:     if(!object->itersators) {
1368:         zend_throw_exception_ex(spl_ce_LogicException, 0,
1369:             "The object is in an invalid state as the parent constructor was not called");
1370:         return;
1371:     }
1372:
1373:     spl_recursive_tree_iterator_current(object);
1374:     /* '}' */
1375:
1376:     /* {{{ proto void RecursiveTreeIterator::key()
1377:  Returns the current key */
1378: SPL_METHOD (RecursiveTreeIterator, key)
1379: {
1380:     spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(getThis());
1381:
1382:     if (zend_parse_parameters_none() == FAILURE) {
1383:         return;
1384:     }
1385:
1386:     if(!object->itersators) {
1387:         zend_throw_exception_ex(spl_ce_LogicException, 0,
1388:             "The object is in an invalid state as the parent constructor was not called");
1389:         return;
1390:     }
1391:
1392:     spl_recursive_tree_iterator_key(object);
1393:     /* '}' */
1394:
1395:     /* {{{ proto void RecursiveTreeIterator::next()
1396:  Move to the next element */
1397: SPL_METHOD (RecursiveTreeIterator, next)
1398: {
1399:     spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(getThis());
1400:
1401:     if (zend_parse_parameters_none() == FAILURE) {
1402:         return;
1403:     }
1404:
1405:     if(!object->itersators) {
1406:         zend_throw_exception_ex(spl_ce_LogicException, 0,
1407:             "The object is in an invalid state as the parent constructor was not called");
1408:         return;
1409:     }
1410:
1411:     spl_recursive_tree_iterator_next(object);
1412:     /* '}' */
1311:
1312:     zval_ptr_dtor(&prefix);
1313:     zval_ptr_dtor(&key);
1314:     zval_ptr_dtor(&postfix);
1315:
1316:     RETURN_NEW_STR(str);
1317:     /* '}' */
1318:
1319:     /* {{{ proto void RecursiveTreeIterator::rewind()
1320:  Rewind the iterator to the first element */
1321: SPL_METHOD (RecursiveTreeIterator, rewind)
1322: {
1323:     spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(getThis());
1324:
1325:     if (zend_parse_parameters_none() == FAILURE) {
1
```

```

1684:     }
1685:
1686: static inline int spi_dual_it_valid(spi_dual_it_object *intern)
1687: {
1688:     if (intern->inner_iterator) {
1689:         return FAILURE;
1690:     }
1691:     /* FAILURE / SUCCESS */
1692:     return intern->inner_iterator->funcs->valid(intern->inner_iterator);
1693: }
1694:
1695: static inline int spi_dual_it_fetch(spi_dual_it_object *intern, int check_more)
1696: {
1697:     sval *data;
1698:
1699:     spi_dual_it_free(intern);
1700:     if (check_more || spi_dual_it_valid(intern) == SUCCESS) {
1701:         data = intern->inner_iterator->funcs->get_current_data(intern->inner_iterator);
1702:         if (data) {
1703:             EVAL_COPY(intern->current_data, data);
1704:         }
1705:
1706:         if (intern->inner_iterator->funcs->get_current_key()
1707:             ? intern->inner_iterator->funcs->get_current_key(intern->inner_iterator, &intern-
1708:             >key(exception)) {
1709:             sval_get_str(intern->current_key);
1710:             EVAL_UNDEF(intern->current_key);
1711:         }
1712:     } else {
1713:         EVAL_LONG(intern->current_key, intern->current_pos);
1714:     }
1715:     return EG(exception) ? FAILURE : SUCCESS;
1716: }
1717:
1718: return FAILURE;
1719: }
1720:
1721: static inline void spi_dual_it_next(spi_dual_it_object *intern, int do_free)
1722: {
1723:     if (do_free) {
1724:         spi_dual_it_free(intern);
1725:     }
1726:     /* if (intern->inner_iterator) {
1727:         zend_throw_error(NULL, "The inner constructor wasn't initialized with an iterator
1728:         return;
1729:     } */
1730:
1731:     intern->inner_iterator->funcs->move_forward(intern->inner_iterator);
1732:     intern->current_pos++;
1733:
1734:     /* (( proto void ParamTIterator::rewind()
1735:         proto void IteratorTIterator::rewind()
1736:         Rewind the iterator
1737:         */
1738:     spi_dual_it_rewind(intern);
1739:
1740:     if (zend_parse_parameters_none() == FAILURE) {
1741:         return;
1742:     }
1743:
1744:     SPI_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
1745:
1746:     spi_dual_it_rewind(intern);
1747:     spi_dual_it_fetch(intern, 1);
1748:     /* */
1749:
1750:     /* (( proto bool FilterTIterator::valid()
1751:         proto bool ParamTIterator::valid()
1752:         proto bool IteratorTIterator::valid()
1753:         proto bool NoValidTIterator::valid()
1754:         Check whether the current element is valid */
1755:     SPI_METHOD(spi_dual_it, valid);
1756:
1757:     spi_dual_it_object *intern;
1758:
1759:     if (zend_parse_parameters_none() == FAILURE) {
1760:         return;
1761:     }
1762:
1763:     SPI_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
1764:
1765:     RETURN_BOOL(Z_TYPE(intern->current_data) != IS_UNDEF);
1766:     /* */
1767:
1768:     /* (( proto mixed FilterTIterator::key()
1769:         proto mixed CachingTIterator::key()
1770:         proto mixed LimitTIterator::key()
1771:         proto mixed ParamTIterator::key()
1772:         proto mixed IteratorTIterator::key()
1773:         proto mixed NoValidTIterator::key()
1774:         proto mixed AppendTIterator::key()
1775:         Get the current key */
1776:     SPI_METHOD(spi_dual_it, key);
1777:
1778:     spi_dual_it_object *intern;
1779:
1780:     if (zend_parse_parameters_none() == FAILURE) {
1781:         return;
1782:     }
1783:
1784:     SPI_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
1785:
1786:     if (Z_TYPE(intern->current_key) != IS_UNDEF) {
1787:         sval *value = intern->current_key;
1788:
1789:         EVAL_DEREF(value);
1790:         EVAL_COPY(return_value, value);
1791:     } else {
1792:         RETURN_NULL();
1793:     }
1794:     /* */
1795:
1796:     /* (( proto mixed FilterTIterator::current()
1797:         proto mixed CachingTIterator::current()
1798:         proto mixed LimitTIterator::current()
1799:         proto mixed ParamTIterator::current()
1800:         proto mixed IteratorTIterator::current()
1801:         proto mixed NoValidTIterator::current()
1802:         Get the current element value */
1803:     SPI_METHOD(spi_dual_it, current);
1804:
1805:     spi_dual_it_object *intern;
1806:
1807:     if (zend_parse_parameters_none() == FAILURE) {
1808:         return;
1809:     }
1810:
1811:     SPI_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
1812:
1813:     if (Z_TYPE(intern->current_data) != IS_UNDEF) {
1814:         sval *value = intern->current_data;
1815:
1816:         EVAL_DEREF(value);
1817:         EVAL_COPY(return_value, value);
1818:     } else {
1819:         RETURN_NULL();
1820:     }
1821:     /* */
1822:
1823:     /* (( proto void ParamTIterator::next()
1824:         proto void IteratorTIterator::next()
1825:         proto void NoValidTIterator::next()
1826:         Move the iterator forward */
1827:     SPI_METHOD(spi_dual_it, next);
1828:
1829:     spi_dual_it_object *intern;
1830:
1831:     if (zend_parse_parameters_none() == FAILURE) {
1832:         return;
1833:     }
1834:
1835:     SPI_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
1836:
1837:     spi_dual_it_next(intern, 1);
1838:     spi_dual_it_fetch(intern, 1);
1839:     /* */
1840:
1841: static inline void spi_filter_it_fetch(sval *this, spi_dual_it_object *intern)
1842: {
1843:     sval *retval;
1844:
1845:     while (spi_dual_it_fetch(intern, 1) == SUCCESS) {
1846:         zend_sval_init_with_3_params(this, intern->std.ow, NULL, "accept", &retval);
1847:         if (Z_TYPE(retval) != IS_UNDEF) {
1848:             if (zend_is_true(retval)) {
1849:                 sval_get_str(&retval);
1850:                 return;
1851:             }
1852:             EVAL_GET_STR(&retval);
1853:         }
1854:         if (EG(exception)) {
1855:             return;
1856:         }
1857:         intern->inner_iterator->funcs->move_forward(intern->inner_iterator);
1858:     }
1859:     spi_dual_it_free(intern);
1860:
1861: static inline void spi_filter_it_rewind(sval *this, spi_dual_it_object *intern)
1862: {
1863:     spi_dual_it_rewind(intern);
1864:     spi_filter_it_fetch(this, intern);
1865: }
1866:
1867: static inline void spi_filter_it_next(sval *this, spi_dual_it_object *intern)
1868: {
1869:     spi_dual_it_next(intern, 1);

```

```

1871:     spl_filter_it_fetch(this, intern);
1872: }
1873:
1874: /* {{{ proto void FilterIterator::rewind()
1875:  * Rewind the iterator */
1876: SPL_METHOD(FilterIterator, rewind)
1877: {
1878:     spl_dual_it_object *intern;
1879:
1880:     IF (zend_parse_parameters_none() == FAILURE) {
1881:         return;
1882:     }
1883:
1884:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
1885:     spl_filter_it_rewind(getThis(), intern);
1886:     /* }}} */
1887:
1888: /* {{{ proto void FilterIterator::next()
1889:  * Move the iterator forward */
1890: SPL_METHOD(FilterIterator, next)
1891: {
1892:     spl_dual_it_object *intern;
1893:
1894:     IF (zend_parse_parameters_none() == FAILURE) {
1895:         return;
1896:     }
1897:
1898:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
1899:     spl_filter_it_next(getThis(), intern);
1900:     /* }}} */
1901:
1902: /* {{{ proto void RecursiveCallbackFilterIterator::__construct(RecursiveIterator it, callback func)
1903:  * Create a RecursiveCallbackFilterIterator from a RecursiveIterator */
1904: SPL_METHOD(RecursiveCallbackFilterIterator, __construct)
1905: {
1906:     spl_dual_it_construct (INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ca_RecursiveCallbackFilterIterator, spl_ca_RecursiveIterator, DIT_RecursiveCallbackFilter
1907: Iterator);
1908:     /* }}} */
1909:
1910: /* {{{ proto void RecursiveFilterIterator::__construct(RecursiveIterator it)
1911:  * Create a RecursiveFilterIterator from a RecursiveIterator */
1912: SPL_METHOD(RecursiveFilterIterator, __construct)
1913: {
1914:     spl_dual_it_construct (INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ca_RecursiveFilterIterator, spl_ca_RecursiveIterator, DIT_RecursiveFilterIterator);
1915:     /* }}} */
1916:
1917: /* {{{ proto bool RecursiveFilterIterator::hasChildren()
1918:  * Check whether the inner iterator's current element has children */
1919: SPL_METHOD(RecursiveFilterIterator, hasChildren)
1920: {
1921:     spl_dual_it_object *intern;
1922:     zval retval;
1923:
1924:     IF (zend_parse_parameters_none() == FAILURE) {
1925:         return;
1926:     }
1927:
1928:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
1929:
1930:     zend_call_method_with_0_params(&intern->inner.object, intern->inner.ce, NULL, "haschildren", &retval);
1931:     IF (Z_TYPE(retval) != IS_UNDEF) {
1932:         RETURN_ZVAL(&retval, 0, 1);
1933:     } else {
1934:         RETURN_FALSE;
1935:     }
1936:     /* }}} */
1937:
1938: /* {{{ proto RecursiveFilterIterator RecursiveFilterIterator::getChildren()
1939:  * Return the inner iterator's children contained in a RecursiveFilterIterator */
1940: SPL_METHOD(RecursiveFilterIterator, getChildren)
1941: {
1942:     spl_dual_it_object *intern;
1943:     zval retval;
1944:
1945:     IF (zend_parse_parameters_none() == FAILURE) {
1946:         return;
1947:     }
1948:
1949:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
1950:
1951:     zend_call_method_with_0_params(&intern->inner.object, intern->inner.ce, NULL, "getchildren", &retval);
1952:     IF (EG(exception) && Z_TYPE(retval) != IS_UNDEF) {
1953:         spl_instantiate_arg_ext(Z_OBJCE_P(getThis()), return_value, &retval);
1954:     }
1955:     zend_ptr_dtor(&retval);
1956:     /* }}} */
1957:
1958: /* {{{ proto RecursiveCallbackFilterIterator RecursiveCallbackFilterIterator::getChildren()
1959:  * Return the inner iterator's children contained in a RecursiveCallbackFilterIterator */
1960: SPL_METHOD(RecursiveCallbackFilterIterator, getChildren)
1961: {
1962:     spl_dual_it_object *intern;
1963:     zval retval;
1964:
1965:     IF (zend_parse_parameters_none() == FAILURE) {
1966:         return;
1967:     }
1968:
1969:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
1970:
1971:     zend_call_method_with_0_params(&intern->inner.object, intern->inner.ce, NULL, "getchildren", &retval);
1972:     IF (EG(exception) && Z_TYPE(retval) != IS_UNDEF) {
1973:         spl_instantiate_arg_ext(Z_OBJCE_P(getThis()), return_value, &retval, &intern->callback->fci.function_name);
1974:     }
1975:     zend_ptr_dtor(&retval);
1976:     /* }}} */
1977:
1978: /* {{{ proto void ParentIterator::__construct(RecursiveIterator it)
1979:  * Create a ParentIterator from a RecursiveIterator */
1980: SPL_METHOD(ParentIterator, __construct)
1981: {
1982:     spl_dual_it_construct (INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ca_ParentIterator, spl_ca_RecursiveIterator, DIT_ParentIterator);
1983:     /* }}} */
1984:
1985: #if HAVE_PCRE || HAVE_UNICODE_PCRE
1986: /* {{{ proto void RegexIterator::__construct(Iterator it, string regex [, int mode [, int flags [, int preg_flags]]])
1987:  * Create an RegexIterator from another iterator and a regular expression */
1988: SPL_METHOD(RegexIterator, __construct)
1989: {
1990:     spl_dual_it_construct (INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ca_RegexIterator, zend_ca_iterator, DIT_RegexIterator);
1991:     /* }}} */
1992:
1993: /* {{{ proto bool CallbackFilterIterator::accept()
1994:  * Call the callback with the current value, the current key and the inner iterator as arguments */
1995: SPL_METHOD(CallbackFilterIterator, accept)
1996: {
1997:     spl_dual_it_object *intern = Z_SPDUAL_IT_P(getThis());
1998:     zend_fcall_info *fci = &intern->callback->fci;
1999:     zend_fcall_info_cache *fci_cache = &intern->callback->fci_cache;
2000:     zval params[3];
2001:
2002:     IF (zend_parse_parameters_none() == FAILURE) {
2003:         return;
2004:     }
2005:
2006:     IF (Z_TYPE(intern->current.data) == IS_UNDEF || Z_TYPE(intern->current.key) == IS_UNDEF) {
2007:         RETURN_FALSE;
2008:     }
2009:
2010:     ZVAL_COPY_VALUE(&params[0], &intern->current.data);
2011:     ZVAL_COPY_VALUE(&params[1], &intern->current.key);
2012:     ZVAL_COPY_VALUE(&params[2], &intern->inner.object);
2013:     fci->retval = return_value;
2014:     fci->param_count = 3;
2015:     fci->params = params;
2016:     fci->no_separation = 0;
2017:
2018:     IF (zend_call_function(fci, fci) != SUCCESS || Z_ISUNDEF(return_value)) {
2019:         RETURN_FALSE;
2020:     }
2021:
2022:     IF (EG(exception)) {
2023:         RETURN_NULL();
2024:     }
2025:
2026:     /* zend_call_function may change args to IS_REF */
2027:     ZVAL_COPY_VALUE(&intern->current.data, &params[0]);
2028:     ZVAL_COPY_VALUE(&intern->current.key, &params[1]);
2029:     /* }}} */
2030:
2031: /* {{{ proto bool RegexIterator::accept()
2032:  * Match (string)current() against regular expression */
2033: SPL_METHOD(RegexIterator, accept)
2034: {
2035:     spl_dual_it_object *intern;
2036:     zend_string *result, *subject;
2037:     zend_uchar count = 0;
2038:     zval account, *replacement, tmp_replacement, rv;
2039:     pcre2_match_data *match_data;
2040:     pcre2_code *re;
2041:     int rc;
2042:
2043:     IF (zend_parse_parameters_none() == FAILURE) {
2044:         return;
2045:     }
2046:
2047:
2048:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2049:
2050:     IF (Z_TYPE(intern->current.data) == IS_UNDEF) {
2051:         RETURN_FALSE;
2052:     }
2053:
2054:     IF (intern->u.regex.flags & REGIT_USE_KEY) {
2055:         subject = zval_get_string(intern->current.key);
2056:     } else {
2057:         IF (Z_TYPE(intern->current.data) == IS_ARRAY) {

```



```

2246: /* {{{ proto RecursiveHashIterator RecursiveHashIterator::__construct()
2247:  * Return the inner iterator's children contained in a RecursiveHashIterator */
2248: SPL_METHOD(RecursiveHashIterator, getChildren)
2249: {
2250:     spl_dual_it_object *intern;
2251:     zval rretval;
2252:
2253:     if (zend_parse_parameters_none() == FAILURE) {
2254:         return;
2255:     }
2256:
2257:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2258:
2259:     zend_call_method_with_0_params(intern->inner->obj, intern->inner->ce, NULL, "getchildren", &rretval);
2260:     if (!EG(exception)) {
2261:         zval args[5];
2262:
2263:         ZVAL_COPY(&args[0], &rretval);
2264:         ZVAL_STR_COPY(&args[1], intern->u.regex.regex);
2265:         ZVAL_LONG(&args[2], intern->u.regex.mode);
2266:         ZVAL_LONG(&args[3], intern->u.regex.flags);
2267:         ZVAL_LONG(&args[4], intern->u.regex.preg_flags);
2268:
2269:         spl_instantiate_arg_n(2, OBJ_CEP, getThis(), return_value, 5, args);
2270:
2271:         zval_ptr_dtor(&args[0]);
2272:         zval_ptr_dtor(&args[1]);
2273:         zval_ptr_dtor(&args[2]);
2274:         zval_ptr_dtor(&args[3]);
2275:     } /* }}} */
2276:
2277: SPL_METHOD(RecursiveHashIterator, accept)
2278: {
2279:     spl_dual_it_object *intern;
2280:
2281:     if (zend_parse_parameters_none() == FAILURE) {
2282:         return;
2283:     }
2284:
2285:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2286:
2287:     if (EG_TUPLE(intern->current.data) == IS_UNDEF) {
2288:         RETURN_FALSE;
2289:     } else if (EG_TUPLE(intern->current.data) == IS_ARRAY) {
2290:         RETURN_BOOL(zend_hash_num_elements(2_ARRAYVAL(intern->current.data)) > 0);
2291:     }
2292:
2293:     zend_call_method_with_0_params(getThis(), spl_ce_RegexIterator, NULL, "accept", return_value);
2294: }
2295:
2296: #endif
2297:
2298: /* {{{ spl_dual_it_dtor */
2299: static void spl_dual_it_dtor(zend_object *object)
2300: {
2301:     spl_dual_it_object *obj = spl_dual_it_from_obj(object);
2302:
2303:     /* call standard dtor */
2304:     zend_objects_destroy_object(obj);
2305:
2306:     spl_dual_it_free(obj);
2307: }
2308:
2309: if (object->inner.iterator) {
2310:     zend_iterator_dtor(object->inner.iterator);
2311: }
2312: /* }}} */
2313:
2314: /* {{{ spl_dual_it_free_storage */
2315: static void spl_dual_it_free_storage(zend_object *object)
2316: {
2317:     spl_dual_it_object *obj = spl_dual_it_from_obj(object);
2318:
2319:     if (!IS_UNDEF(object->inner.obj)) {
2320:         zval_ptr_dtor(object->inner.obj);
2321:     }
2322:
2323:     if (object->dit_type == DIT_AppendIterator) {
2324:         zend_iterator_dtor(object->u.append.iterator);
2325:     }
2326:     if (EG_TUPLE(object->u.append.array) != IS_UNDEF) {
2327:         zval_ptr_dtor(object->u.append.array);
2328:     }
2329: }
2330:
2331: if (object->dit_type == DIT_CachingIterator) {
2332:     zval_ptr_dtor(object->u.caching.cache);
2333: }
2334:
2335: if (HAVE_PCRE || HAVE_BUNDLED_PCRE)
2336:     if (object->dit_type == DIT_RegexIterator || object->dit_type == DIT_RecursiveRegexIterator) {
2337:         if (object->u.regex.pos) {
2338:             pcre_pos_decref(object->u.regex.pos);
2339:         }
2340:         if (object->u.regex.regex) {
2341:             zend_string_release(object->u.regex.regex);
2342:         }
2343:     }
2344: #endif
2345:
2346: if (object->dit_type == DIT_CallbackFilterIterator) {
2347:     if (object->u.cbfilter) {
2348:         spl_dual_it_from_obj(cbfilter = object->u.cbfilter);
2349:         object->u.cbfilter = NULL;
2350:         zval_ptr_dtor(cbfilter->fci.function_name);
2351:         if (cbfilter->fci.obj) {
2352:             OBJ_RELEASE(cbfilter->fci.obj);
2353:         }
2354:         zfree(cbfilter);
2355:     }
2356: }
2357:
2358: zend_object_std_dtor(object->std);
2359: }
2360: /* }}} */
2361:
2362: /* {{{ spl_dual_it_new */
2363: static zend_object *spl_dual_it_new(zend_class_entry *class_type)
2364: {
2365:     spl_dual_it_object *intern;
2366:
2367:     intern = zend_object_alloc(sizeof(spl_dual_it_object), class_type);
2368:     intern->dit_type = DIT_Unknown;
2369:
2370:     zend_object_std_init(intern->std, class_type);
2371:     object_properties_init(intern->std, class_type);
2372:
2373:     intern->std.handlers = spl_handlers_dual_it;
2374:     return intern->std;
2375: }
2376: /* }}} */
2377:
2378: ZEND_BEGIN_ARG_INFO(arginfo_filter_it_construct, 0)
2379:     ZEND_ARG_OBJ_INFO(0, iterator, Iterator, 0)
2380: ZEND_END_ARG_INFO();
2381:
2382: static const zend_function_entry spl_funcs_FilterIterator[] = {
2383:     SPL_ME(FilterIterator, __construct, arginfo_filter_it_construct, ZEND_ACC_PUBLIC)
2384:     SPL_ME(FilterIterator, rewind, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2385:     SPL_ME(dual_it, valid, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2386:     SPL_ME(dual_it, key, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2387:     SPL_ME(dual_it, current, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2388:     SPL_ME(FilterIterator, next, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2389:     SPL_ME(dual_it, getInnerIterator, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2390:     SPL_ABSTRACT_ME(FilterIterator, accept, arginfo_recursive_it_void)
2391:     PHP_FE_END
2392: };
2393:
2394: ZEND_BEGIN_ARG_INFO(arginfo_callback_filter_it_construct, 0)
2395:     ZEND_ARG_OBJ_INFO(0, iterator, Iterator, 0)
2396:     ZEND_ARG_INFO(0, callback)
2397: ZEND_END_ARG_INFO();
2398:
2399: static const zend_function_entry spl_funcs_CallbackFilterIterator[] = {
2400:     SPL_ME(CallbackFilterIterator, __construct, arginfo_callback_filter_it_construct, ZEND_ACC_PUBLIC)
2401:     SPL_ME(CallbackFilterIterator, accept, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2402:     PHP_FE_END
2403: };
2404:
2405: ZEND_BEGIN_ARG_INFO(arginfo_recursive_callback_filter_it_construct, 0)
2406:     ZEND_ARG_OBJ_INFO(0, iterator, RecursiveIterator, 0)
2407:     ZEND_ARG_INFO(0, callback)
2408: ZEND_END_ARG_INFO();
2409:
2410: static const zend_function_entry spl_funcs_RecursiveCallbackFilterIterator[] = {
2411:     SPL_ME(RecursiveCallbackFilterIterator, __construct, arginfo_recursive_callback_filter_it_construct, ZEND_ACC_PUBLIC)
2412:     SPL_ME(RecursiveCallbackFilterIterator, hasChildren, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2413:     SPL_ME(RecursiveCallbackFilterIterator, getChildren, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2414:     PHP_FE_END
2415: };
2416:
2417: ZEND_BEGIN_ARG_INFO(arginfo_parent_it_construct, 0)
2418:     ZEND_ARG_OBJ_INFO(0, iterator, RecursiveIterator, 0)
2419: ZEND_END_ARG_INFO();
2420:
2421: static const zend_function_entry spl_funcs_RecursiveParentIterator[] = {
2422:     SPL_ME(RecursiveParentIterator, __construct, arginfo_parent_it_construct, ZEND_ACC_PUBLIC)
2423:     SPL_ME(RecursiveParentIterator, accept, RecursiveFilterIterator, hasChildren, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2424:     PHP_FE_END
2425: };
2426:
2427:
2428: static const zend_function_entry spl_funcs_ParentIterator[] = {
2429:     SPL_ME(ParentIterator, __construct, arginfo_parent_it_construct, ZEND_ACC_PUBLIC)
2430:     SPL_ME(ParentIterator, accept, RecursiveFilterIterator, hasChildren, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2431:     PHP_FE_END
2432: };
2433:
2434:
2435: #if HAVE_PCRE || HAVE_BUNDLED_PCRE
2436:     ZEND_BEGIN_ARG_INFO_EX(arginfo_regex_it_construct, 0, 0, 2)
2437:         ZEND_ARG_OBJ_INFO(0, iterator, Iterator, 0)
2438:         ZEND_ARG_INFO(0, regex)
2439:         ZEND_ARG_INFO(0, mode)
2440:         ZEND_ARG_INFO(0, flags)
2441:         ZEND_ARG_INFO(0, preg_flags)
2442:     ZEND_END_ARG_INFO();
2443:
2444:     ZEND_BEGIN_ARG_INFO_EX(arginfo_regex_it_set_mode, 0, 0, 1)
2445:         ZEND_ARG_INFO(0, mode)
2446:     ZEND_END_ARG_INFO();
2447:
2448:     ZEND_BEGIN_ARG_INFO_EX(arginfo_regex_it_set_preg, 0, 0, 1)
2449:         ZEND_ARG_INFO(0, flags)
2450:     ZEND_END_ARG_INFO();
2451:
2452:     ZEND_BEGIN_ARG_INFO_EX(arginfo_regex_it_set_preg_flags, 0, 0, 1)
2453:         ZEND_ARG_INFO(0, preg_flags)
2454:     ZEND_END_ARG_INFO();
2455:
2456:     static const zend_function_entry spl_funcs_RegexIterator[] = {
2457:         SPL_ME(RegexIterator, __construct, arginfo_regex_it_construct, ZEND_ACC_PUBLIC)
2458:         SPL_ME(RegexIterator, accept, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2459:         SPL_ME(RegexIterator, getMode, arginfo_regex_it_set_mode, ZEND_ACC_PUBLIC)
2460:         SPL_ME(RegexIterator, getFlags, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2461:         SPL_ME(RegexIterator, getFlags, arginfo_regex_it_set_flags, ZEND_ACC_PUBLIC)
2462:         SPL_ME(RegexIterator, getFlags, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2463:         SPL_ME(RegexIterator, setFlags, arginfo_regex_it_set_preg_flags, ZEND_ACC_PUBLIC)
2464:         SPL_ME(RegexIterator, getRegex, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2465:         PHP_FE_END
2466:     };
2467:
2468:     ZEND_BEGIN_ARG_INFO_EX(arginfo_rec_regex_it_construct, 0, 0, 2)
2469:         ZEND_ARG_OBJ_INFO(0, iterator, RecursiveIterator, 0)
2470:         ZEND_ARG_INFO(0, regex)
2471:         ZEND_ARG_INFO(0, mode)
2472:         ZEND_ARG_INFO(0, flags)
2473:         ZEND_ARG_INFO(0, preg_flags)
2474:     ZEND_END_ARG_INFO();
2475:
2476:     static const zend_function_entry spl_funcs_RecursiveRegexIterator[] = {
2477:         SPL_ME(RecursiveRegexIterator, __construct, arginfo_rec_regex_it_construct, ZEND_ACC_PUBLIC)
2478:         SPL_ME(RecursiveRegexIterator, accept, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2479:         SPL_ME(RecursiveRegexIterator, hasChildren, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2480:         SPL_ME(RecursiveRegexIterator, getChildren, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2481:         PHP_FE_END
2482:     };
2483: #endif
2484:
2485: static inline int spl_limit_it_valid(spl_dual_it_object *intern)
2486: {
2487:     /* FAILURE / SUCCESS */
2488:     if (intern->u.limit.count != -1 && intern->current.pos >= intern->u.limit.offset + intern->u.limit.count) {
2489:         return FAILURE;
2490:     } else {
2491:         return spl_dual_it_valid(intern);
2492:     }
2493: }
2494:
2495: static inline void spl_limit_it_seek(spl_dual_it_object *intern, zend_long pos)
2496: {
2497:     zval zpos;
2498:
2499:     spl_dual_it_free(intern);
2500:     if (pos < intern->u.limit.offset) {
2501:         zend_throw_exception_ex(spl_ce_OutOfBoundsException, 0, "Cannot seek to " ZEND_LONG_FMT " which is below the offset " ZEND_LONG_FMT, pos, intern->u.limit.offset);
2502:     }
2503:
2504:     if (pos >= intern->u.limit.offset + intern->u.limit.count && intern->u.limit.count != -1) {
2505:         zend_throw_exception_ex(spl_ce_OutOfBoundsException, 0, "Cannot seek to " ZEND_LONG_FMT " which is behind offset " ZEND_LONG_FMT " plus count " ZEND_LONG_FMT, pos, intern->u.limit.offset, intern->u.limit.count);
2506:     }
2507:
2508:     if (pos != intern->current.pos && instanceof_function(intern->inner->ce, spl_ce_SeekableIterator)) {
2509:         ZVAL_LONG(&zpos, pos);
2510:         zend_call_method_with_1_params(intern->inner->obj, intern->inner->ce, NULL, "seek", NULL, &zpos);
2511:         if (!EG(exception)) {
2512:             zval_ptr_dtor(&zpos);
2513:             intern->current.pos = pos;
2514:             if (spl_limit_it_valid(intern) == SUCCESS) {
2515:                 spl_dual_it_fetch(intern, 0);
2516:             }
2517:         }
2518:     } else {
2519:         /* emulate the forward seek, by next() calls */
2520:         /* a back ward seek is done by a previous rewind() */
2521:         if (pos < intern->current.pos) {
2522:             spl_dual_it_rewind(intern);
2523:         }
2524:
2525:         while (pos > intern->current.pos && spl_limit_it_valid(intern) == SUCCESS) {
2526:             spl_dual_it_next(intern, 1);
2527:         }
2528:         if (spl_limit_it_valid(intern) == SUCCESS) {
2529:             spl_dual_it_fetch(intern, 1);
2530:         }
2531:     }
2532: }
2533:
2534: /* {{{ proto void LimitIterator::__construct(Iterator $i, int offset, int count)
2535:  * Construct a LimitIterator from an Iterator with a given starting offset and optionally a maximum count */
2536: SPL_METHOD(LimitIterator, __construct)
2537: {
2538:     spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ce_LimitIterator, zend_ce_iterator, DIT_LimitIterator);
2539: } /* }}} */
2540:
2541: /* {{{ proto void LimitIterator::rewind()
2542:  * Rewind the iterator to the specified starting offset */
2543: SPL_METHOD(LimitIterator, rewind)
2544: {
2545:     spl_dual_it_object *intern;
2546:
2547:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2548:     spl_dual_it_rewind(intern);
2549:     spl_limit_it_seek(intern, intern->u.limit.offset);
2550: } /* }}} */
2551:
2552: /* {{{ proto bool LimitIterator::valid()
2553:  * Check whether the current element is valid */
2554: SPL_METHOD(LimitIterator, valid)
2555: {
2556:     spl_dual_it_object *intern;
2557:
2558:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2559:
2560:     /* RETURN_BOOL(spl_limit_it_valid(intern) == SUCCESS); */
2561:     RETURN_BOOL(intern->u.limit.count == -1 || intern->current.pos < intern->u.limit.offset + intern->u.limit.count && EG_TUPLE(intern->current.data) != IS_UNDEF);
2562: } /* }}} */
2563:
2564: /* {{{ proto void LimitIterator::next()
2565:  * Move the iterator forward */
2566: SPL_METHOD(LimitIterator, next)
2567: {
2568:     spl_dual_it_object *intern;
2569:
2570:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2571:
2572:     spl_dual_it_next(intern, 1);
2573:     if (intern->u.limit.count == -1 || intern->current.pos < intern->u.limit.offset + intern->u.limit.count) {
2574:         spl_dual_it_fetch(intern, 1);
2575:     }
2576: } /* }}} */
2577:
2578: /* {{{ proto void LimitIterator::seek(int position)
2579:  * Seek to the given position */
2580: SPL_METHOD(LimitIterator, seek)
2581: {
2582:     spl_dual_it_object *intern;
2583:     zend_long pos;
2584:
2585:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "l", &pos) == FAILURE) {
2586:         return;
2587:     }
2588:
2589:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2590:     spl_limit_it_seek(intern, pos);
2591:     RETURN_LONG(intern->current.pos);
2592: } /* }}} */
2593:
2594: /* {{{ proto int LimitIterator::getPosition()
2595:  * Return the current position */
2596: SPL_METHOD(LimitIterator, getPosition)
2597: {
2598:     spl_dual_it_object *intern;
2599:
2600:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2601:     RETURN_LONG(intern->current.pos);
2602: } /* }}} */
2603:
2604: ZEND_BEGIN_ARG_INFO(arginfo_seekable_it_seek, 0)
2605:     ZEND_ARG_INFO(0, position)
2606: ZEND_END_ARG_INFO();
2607:
2608: static const zend_function_entry spl_funcs_SeekableIterator[] = {
2609:     SPL_ABSTRACT_ME(SeekableIterator, seek, arginfo_seekable_it_seek)
2610:     PHP_FE_END
2611: };
2612:
2613: ZEND_BEGIN_ARG_INFO_EX(arginfo_limit_it_construct, 0, 0, 1)
2614:     ZEND_ARG_OBJ_INFO(0, iterator, Iterator, 0)
2615:     ZEND_ARG_INFO(0, offset)
2616:     ZEND_ARG_INFO(0, count)
2617: ZEND_END_ARG_INFO();
2618:
2619: ZEND_BEGIN_ARG_INFO_EX(arginfo_limit_it_seek, 0)

```

```

2619: ZEND_ARG_INFO(0, position)
2620: ZEND_END_ARG_INFO();
2621:
2622: static const zend_function_entry spl_funcs_limitIterator[] = {
2623:     SPL_ME(limitIterator, __construct, arginfo_limit_it__construct, ZEND_ACC_PUBLIC)
2624:     SPL_ME(limitIterator, rewind, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2625:     SPL_ME(limitIterator, valid, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2626:     SPL_ME(dual_it, key, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2627:     SPL_ME(dual_it, current, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2628:     SPL_ME(limitIterator, next, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2629:     SPL_ME(limitIterator, seek, arginfo_limit_it_seek, ZEND_ACC_PUBLIC)
2630:     SPL_ME(limitIterator, getPosition, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2631:     SPL_ME(dual_it, getInnerIterator, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2632:     PHP_FE_END
2633: };
2634:
2635: static inline int spl_caching_it_valid(spl_dual_it_object *intern)
2636: {
2637:     return intern->u.caching.flags & CIT_VALID ? SUCCESS : FAILURE;
2638: }
2639:
2640: static inline int spl_caching_it_has_next(spl_dual_it_object *intern)
2641: {
2642:     return spl_dual_it_valid(intern);
2643: }
2644:
2645: static inline void spl_caching_it_next(spl_dual_it_object *intern)
2646: {
2647:     if (spl_dual_it_fetch(intern, 1) == SUCCESS) {
2648:         intern->u.caching.flags |= CIT_VALID;
2649:         /* Full cache ? */
2650:         if (intern->u.caching.flags & CIT_FULL_CACHE) {
2651:             zval *key = sintern->current.key;
2652:             zval *data = sintern->current.data;
2653:
2654:             ZVAL_DEREF(data);
2655:             Z_TRY_ADDREF_P(data);
2656:             array_wat_eval_key(Z_ARRVAL(intern->u.caching.cache), key, data);
2657:             zval_ptr_dtor(data);
2658:         }
2659:         /* Recursion ? */
2660:         if (intern->dit_type == DIT_RecursiveCachingIterator) {
2661:             zval retval, zchildren, zflags;
2662:             zend_call_method_with_0_params(&intern->inner.subject, intern->inner.ce, NULL, "haschildren", &retval);
2663:             if (EG(exception)) {
2664:                 zval_ptr_dtor(&retval);
2665:                 if (intern->u.caching.flags & CIT_CATCH_GET_CHILD) {
2666:                     zend_clear_exception();
2667:                 } else {
2668:                     return;
2669:                 }
2670:             } else {
2671:                 if (zend_is_true(retval)) {
2672:                     zend_call_method_with_0_params(&intern->inner.subject, intern->inner.ce, NULL, "getchildren", &zchildren);
2673:                     if (EG(exception)) {
2674:                         zval_ptr_dtor(&zchildren);
2675:                         if (intern->u.caching.flags & CIT_CATCH_GET_CHILD) {
2676:                             zend_clear_exception();
2677:                         } else {
2678:                             return;
2679:                         }
2680:                     } else {
2681:                         zval_ptr_dtor(&retval);
2682:                         ZVAL_DEREF(zchildren);
2683:                         spl_instantiate_arg_ex2(spl_cw_RecursiveCachingIterator, &intern->u.caching.zchildren, &zchildren, &zflags);
2684:                         zval_ptr_dtor(&zchildren);
2685:                     }
2686:                 }
2687:                 zval_ptr_dtor(&retval);
2688:                 if (EG(exception)) {
2689:                     if (intern->u.caching.flags & CIT_CATCH_GET_CHILD) {
2690:                         zend_clear_exception();
2691:                     } else {
2692:                         return;
2693:                     }
2694:                 }
2695:             }
2696:         }
2697:         if (intern->u.caching.flags & CIT_TOSTRING_USE_INNER | CIT_CALL_TOSTRING) {
2698:             int use_copy;
2699:             zval expc_copy;
2700:             if (intern->u.caching.flags & CIT_TOSTRING_USE_INNER) {
2701:                 ZVAL_COPY_VALUE(&intern->u.caching.zstr, sintern->inner.subject);
2702:             } else {
2703:                 ZVAL_COPY_VALUE(&intern->u.caching.zstr, sintern->current.data);
2704:             }
2705:             use_copy = zend_make_printable_zval(&intern->u.caching.zstr, &expc_copy);
2706:             if (use_copy) {
2707:                 ZVAL_COPY_VALUE(&intern->u.caching.zstr, &expc_copy);
2708:             } else {
2709:                 Z_TRY_ADDREF(intern->u.caching.zstr);
2710:             }
2711:         }
2712:         spl_dual_it_next(intern, 0);
2713:     } else {
2714:         intern->u.caching.flags &= ~CIT_VALID;
2715:     }
2716: }
2717:
2718: static inline void spl_caching_it_rewind(spl_dual_it_object *intern)
2719: {
2720:     spl_dual_it_rewind(intern);
2721:     zend_hash_clean(Z_ARRVAL(intern->u.caching.cache));
2722:     spl_caching_it_next(intern);
2723: }
2724:
2725: /* {{{ proto void CachingIterator::__construct (iterator $it [, $flags = CIT_CALL_TOSTRING])
2726: Construct a CachingIterator from an Iterator */
2727: SPL_METHOD(CachingIterator, __construct)
2728: {
2729:     spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_cw_CachingIterator, zend_cw_iterator, DIT_CachingIterator);
2730: } /* }}} */
2731:
2732: /* {{{ proto void CachingIterator::rewind()
2733: Rewind the iterator */
2734: SPL_METHOD(CachingIterator, rewind)
2735: {
2736:     spl_dual_it_object *intern;
2737:
2738:     if (zend_parse_parameters_none() == FAILURE) {
2739:         return;
2740:     }
2741:
2742:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2743:
2744:     spl_caching_it_rewind(intern);
2745: } /* }}} */
2746:
2747: /* {{{ proto bool CachingIterator::valid()
2748: Check whether the current element is valid */
2749: SPL_METHOD(CachingIterator, valid)
2750: {
2751:     spl_dual_it_object *intern;
2752:
2753:     if (zend_parse_parameters_none() == FAILURE) {
2754:         return;
2755:     }
2756:
2757:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2758:
2759:     RETURN_BOOL(spl_caching_it_valid(intern) == SUCCESS);
2760: } /* }}} */
2761:
2762: /* {{{ proto void CachingIterator::next()
2763: Move the iterator forward */
2764: SPL_METHOD(CachingIterator, next)
2765: {
2766:     spl_dual_it_object *intern;
2767:
2768:     if (zend_parse_parameters_none() == FAILURE) {
2769:         return;
2770:     }
2771:
2772:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2773:
2774:     spl_caching_it_next(intern);
2775: } /* }}} */
2776:
2777: /* {{{ proto bool CachingIterator::hasNext()
2778: Check whether the inner iterator has a valid next element */
2779: SPL_METHOD(CachingIterator, hasNext)
2780: {
2781:     spl_dual_it_object *intern;
2782:
2783:     if (zend_parse_parameters_none() == FAILURE) {
2784:         return;
2785:     }
2786:
2787:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2788:
2789:     RETURN_BOOL(spl_caching_it_has_next(intern) == SUCCESS);
2790: } /* }}} */
2791:
2792: /* {{{ proto string CachingIterator::__toString()
2793: Return the string representation of the current element */
2794: SPL_METHOD(CachingIterator, __toString)
2795: {
2796:     spl_dual_it_object *intern;
2797:
2798:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2799:
2800:     if ((intern->u.caching.flags & CIT_CALL_TOSTRING | CIT_TOSTRING_USE_CURRENT | CIT_TOSTRING_USE_INNER)) {
2801:         zend_throw_exception_ex(spl_cw_BadMethodCallException, 0, "%s does not fetch string value (see CachingIterator::__construct)", ZSTR_VAL(Z_OBJCE_P(getThis()))->name);
2802:         return;
2803:     }
2804:     if (intern->u.caching.flags & CIT_TOSTRING_USE_KEY) {
2805:         ZVAL_COPY(&return_value, sintern->current.key);
2806:     }
2807:     convert_to_string(return_value);
2808:     return;
2809: } else if (intern->u.caching.flags & CIT_TOSTRING_USE_CURRENT) {
2810:     ZVAL_COPY(&return_value, sintern->current.data);
2811:     return;
2812: }
2813: if (Z_TYPE(intern->u.caching.zstr) == IS_STRING) {
2814:     RETURN_STR_COPY(Z_STR_P(intern->u.caching.zstr));
2815: } else {
2816:     RETURN_EMPTY_STRING();
2817: }
2818: } /* }}} */
2819:
2820: /* {{{ proto void CachingIterator::offsetSet (mixed $index, mixed $newval)
2821: Set given index in cache */
2822: SPL_METHOD(CachingIterator, offsetSet)
2823: {
2824:     spl_dual_it_object *intern;
2825:     zend_string *key;
2826:     zval *value;
2827:
2828:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2829:
2830:     if ((intern->u.caching.flags & CIT_FULL_CACHE)) {
2831:         zend_throw_exception_ex(spl_cw_BadMethodCallException, 0, "%s does not use a full cache (see CachingIterator::__construct)", ZSTR_VAL(Z_OBJCE_P(getThis()))->name);
2832:         return;
2833:     }
2834:
2835:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "Ss", &key, &value) == FAILURE) {
2836:         return;
2837:     }
2838:
2839:     Z_TRY_ADDREF_P(value);
2840:     zend_symtable_update(Z_ARRVAL(intern->u.caching.cache), key, value);
2841: }
2842: /* }}} */
2843:
2844: /* {{{ proto string CachingIterator::offsetGet (mixed $index)
2845: Return the internal cache if used */
2846: SPL_METHOD(CachingIterator, offsetGet)
2847: {
2848:     spl_dual_it_object *intern;
2849:     zend_string *key;
2850:     zval *value;
2851:
2852:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2853:
2854:     if ((intern->u.caching.flags & CIT_FULL_CACHE)) {
2855:         zend_throw_exception_ex(spl_cw_BadMethodCallException, 0, "%s does not use a full cache (see CachingIterator::__construct)", ZSTR_VAL(Z_OBJCE_P(getThis()))->name);
2856:         return;
2857:     }
2858:
2859:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "S", &key) == FAILURE) {
2860:         return;
2861:     }
2862:
2863:     if ((value = zend_symtable_find(Z_ARRVAL(intern->u.caching.cache), key)) == NULL) {
2864:         zend_error(E_NOTICE, "Undefined index: %s", ZSTR_VAL(key));
2865:         return;
2866:     }
2867:
2868:     ZVAL_DEREF(value);
2869:     ZVAL_COPY(&return_value, value);
2870: }
2871: /* }}} */
2872:
2873: /* {{{ proto void CachingIterator::offsetUnset (mixed $index)
2874: Unset given index in cache */
2875: SPL_METHOD(CachingIterator, offsetUnset)
2876: {
2877:     spl_dual_it_object *intern;
2878:     zend_string *key;
2879:
2880:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2881:
2882:     if ((intern->u.caching.flags & CIT_FULL_CACHE)) {
2883:         zend_throw_exception_ex(spl_cw_BadMethodCallException, 0, "%s does not use a full cache (see CachingIterator::__construct)", ZSTR_VAL(Z_OBJCE_P(getThis()))->name);
2884:         return;
2885:     }
2886:
2887:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "S", &key) == FAILURE) {
2888:         return;
2889:     }
2890:
2891:     zend_symtable_del(Z_ARRVAL(intern->u.caching.cache), key);
2892: }
2893: /* }}} */
2894:
2895: /* {{{ proto bool CachingIterator::offsetExists (mixed $index)
2896: Return whether the requested index exists */
2897: SPL_METHOD(CachingIterator, offsetExists)
2898: {
2899:     spl_dual_it_object *intern;
2900:     zend_string *key;
2901:
2902:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2903:
2904:     if ((intern->u.caching.flags & CIT_FULL_CACHE)) {
2905:         zend_throw_exception_ex(spl_cw_BadMethodCallException, 0, "%s does not use a full cache (see CachingIterator::__construct)", ZSTR_VAL(Z_OBJCE_P(getThis()))->name);
2906:         return;
2907:     }
2908:
2909:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "S", &key) == FAILURE) {
2910:         return;
2911:     }
2912:
2913:     RETURN_BOOL(zend_symtable_exists(Z_ARRVAL(intern->u.caching.cache), key));
2914: }
2915: /* }}} */
2916:
2917: /* {{{ proto bool CachingIterator::getCache()
2918: Return the cache */
2919: SPL_METHOD(CachingIterator, getCache)
2920: {
2921:     spl_dual_it_object *intern;
2922:
2923:     if (zend_parse_parameters_none() == FAILURE) {
2924:         return;
2925:     }
2926:
2927:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2928:
2929:     if ((intern->u.caching.flags & CIT_FULL_CACHE)) {
2930:         zend_throw_exception_ex(spl_cw_BadMethodCallException, 0, "%s does not use a full cache (see CachingIterator::__construct)", ZSTR_VAL(Z_OBJCE_P(getThis()))->name);
2931:         return;
2932:     }
2933:
2934:     ZVAL_COPY(&return_value, &intern->u.caching.cache);
2935: }
2936: /* }}} */
2937:
2938: /* {{{ proto int CachingIterator::getFlags()
2939: Return the internal flags */
2940: SPL_METHOD(CachingIterator, getFlags)
2941: {
2942:     spl_dual_it_object *intern;
2943:
2944:     if (zend_parse_parameters_none() == FAILURE) {
2945:         return;
2946:     }
2947:
2948:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2949:
2950:     RETURN_LONG(intern->u.caching.flags);
2951: }
2952: /* }}} */
2953:
2954: /* {{{ proto void CachingIterator::setFlags (int $flags)
2955: Set the internal flags */
2956: SPL_METHOD(CachingIterator, setFlags)
2957: {
2958:     spl_dual_it_object *intern;
2959:     zend_long flags;
2960:
2961:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2962:
2963:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "l", &flags) == FAILURE) {
2964:         return;
2965:     }
2966:
2967:     if (spl_cit_check_flags(flags) != SUCCESS) {
2968:         zend_throw_exception(spl_cw_InvalidArgumentException, "Flags must contain only one of CALL_TOSTRING, TOSTRING_USE_KEY, TOSTRING_USE_CURRENT, TOSTRING_USE_INNER", 0);
2969:         return;
2970:     }
2971:
2972:     if ((intern->u.caching.flags & CIT_CALL_TOSTRING) != 0 && (flags & CIT_CALL_TOSTRING) == 0) {
2973:         zend_throw_exception(spl_cw_InvalidArgumentException, "Unsetting flag CALL_TOSTRING is not possible", 0);
2974:         return;
2975:     }
2976:
2977:     if ((intern->u.caching.flags & CIT_TOSTRING_USE_INNER) != 0 && (flags & CIT_TOSTRING_USE_INNER) == 0) {
2978:         zend_throw_exception(spl_cw_InvalidArgumentException, "Unsetting flag TOSTRING_USE_INNER is not possible", 0);
2979:         return;
2980:     }
2981:
2982:     /* clear old (reusable) */
2983:     intern->u.caching.flags = (intern->u.caching.flags & ~CIT_PUBLIC) | (flags & CIT_PUBLIC);
2984: }
2985: /* }}} */
2986:
2987: /* {{{ proto void CachingIterator::count()

```



```

2989: // Number of cached elements */
2990: SPL_METHOD(CachingIterator, count)
2991: {
2992:     spl_dual_it_obj_t *intern;
2993:     if (zend_parse_parameters_none() == FAILURE) {
2994:         return;
2995:     }
2996:
2997:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2998:
2999:     if (! (intern->u.caching.flags & CIT_FULL_CACHE)) {
3000:         zend_throw_exception(spl_ce_BadMethodCallException, 0, "Is does not use a full cache (see CachingIterator::__construct)", ZSTR_VAL(Z_STRCP_P(getThis())->name));
3001:         return;
3002:     }
3003:
3004:     RETURN_LONG(zend_hash_num_elements(Z_ARRVAL(intern->u.caching.cache)));
3005: }
3006: /* }}} */
3007:
3008: ZEND_BEGIN_ARG_INFO_EX(arginfo_caching_it___construct, 0, 0, 1)
3009:     ZEND_ARG_OBJ_INFO(0, iterator, Iterator, 0)
3010:     ZEND_ARG_INFO(0, flags)
3011: ZEND_END_ARG_INFO();
3012:
3013: ZEND_BEGIN_ARG_INFO(arginfo_caching_it_setFlags, 0)
3014:     ZEND_ARG_INFO(0, flags)
3015: ZEND_END_ARG_INFO();
3016:
3017: ZEND_BEGIN_ARG_INFO(arginfo_caching_it_offsetGet, 0)
3018:     ZEND_ARG_INFO(0, index)
3019: ZEND_END_ARG_INFO();
3020:
3021: ZEND_BEGIN_ARG_INFO(arginfo_caching_it_offsetSet, 0)
3022:     ZEND_ARG_INFO(0, index)
3023:     ZEND_ARG_INFO(0, newval)
3024: ZEND_END_ARG_INFO();
3025:
3026: static const zend_function_entry spl_func_CachingIterator[] = {
3027:     SPL_ME(CachingIterator, __construct, arginfo_caching_it___construct, ZEND_ACC_PUBLIC)
3028:     SPL_ME(CachingIterator, rewind, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3029:     SPL_ME(CachingIterator, valid, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3030:     SPL_ME(dual_it, key, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3031:     SPL_ME(dual_it, current, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3032:     SPL_ME(CachingIterator, next, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3033:     SPL_ME(CachingIterator, hasNext, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3034:     SPL_ME(CachingIterator, __toString, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3035:     SPL_ME(dual_it, getInnerIterator, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3036:     SPL_ME(CachingIterator, getFlags, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3037:     SPL_ME(CachingIterator, setFlags, arginfo_caching_it_setFlags, ZEND_ACC_PUBLIC)
3038:     SPL_ME(CachingIterator, offsetGet, arginfo_caching_it_offsetGet, ZEND_ACC_PUBLIC)
3039:     SPL_ME(CachingIterator, offsetSet, arginfo_caching_it_offsetSet, ZEND_ACC_PUBLIC)
3040:     SPL_ME(CachingIterator, offsetUnset, arginfo_caching_it_offsetGet, ZEND_ACC_PUBLIC)
3041:     SPL_ME(CachingIterator, offsetExists, arginfo_caching_it_offsetGet, ZEND_ACC_PUBLIC)
3042:     SPL_ME(CachingIterator, getCache, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3043:     SPL_ME(CachingIterator, count, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3044:     PHP_FE_END
3045: };
3046:
3047: /* {{{ proto void RecursiveCachingIterator::__construct(RecursiveIterator it, flags = CIT_CALL_TO_STRING)
3048:  * Create an iterator from a RecursiveIterator */
3049: SPL_METHOD(RecursiveCachingIterator, __construct)
3050: {
3051:     spl_dual_it_construct (INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ce_RecursiveCachingIterator, spl_ce_RecursiveIterator, DIT_RecursiveCachingIterator);
3052: } /* }}} */
3053:
3054: /* {{{ proto bool RecursiveCachingIterator::hasChildren()
3055:  * Check whether the current element of the inner iterator has children */
3056: SPL_METHOD(RecursiveCachingIterator, hasChildren)
3057: {
3058:     spl_dual_it_obj_t *intern;
3059:
3060:     if (zend_parse_parameters_none() == FAILURE) {
3061:         return;
3062:     }
3063:
3064:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3065:
3066:     RETURN_BOOL(! (TYPE(intern->u.caching.children) != IS_UNDEF));
3067: } /* }}} */
3068:
3069: /* {{{ proto RecursiveCachingIterator RecursiveCachingIterator::getChildren()
3070:  * Return the inner iterator's children as a RecursiveCachingIterator */
3071: SPL_METHOD(RecursiveCachingIterator, getChildren)
3072: {
3073:     spl_dual_it_obj_t *intern;
3074:
3075:     if (zend_parse_parameters_none() == FAILURE) {
3076:         return;
3077:     }
3078:
3079:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3080:
3081:     if (Z_TYPE(intern->u.caching.children) != IS_UNDEF) {
3082:         zval *value = &intern->u.caching.children;
3083:
3084:         ZVAL_DEREF(value);
3085:         ZVAL_COPY(&return_value, value);
3086:     } else {
3087:         RETURN_NULL();
3088:     }
3089: } /* }}} */
3090:
3091: ZEND_BEGIN_ARG_INFO_EX(arginfo_caching_rec_it___construct, 0, ZEND_RETURN_VALUE, 1)
3092:     ZEND_ARG_OBJ_INFO(0, iterator, Iterator, 0)
3093:     ZEND_ARG_INFO(0, flags)
3094: ZEND_END_ARG_INFO();
3095:
3096: static const zend_function_entry spl_func_RecursiveCachingIterator[] = {
3097:     SPL_ME(RecursiveCachingIterator, __construct, arginfo_caching_rec_it___construct, ZEND_ACC_PUBLIC)
3098:     SPL_ME(RecursiveCachingIterator, hasChildren, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3099:     SPL_ME(RecursiveCachingIterator, getChildren, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3100:     PHP_FE_END
3101: };
3102:
3103: /* {{{ proto void IteratorIterator::__construct(Traversable it)
3104:  * Create an iterator from anything that is traversable */
3105: SPL_METHOD(IteratorIterator, __construct)
3106: {
3107:     spl_dual_it_construct (INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ce_IteratorIterator, zend_ce_traversable, DIT_IteratorIterator);
3108: } /* }}} */
3109:
3110: ZEND_BEGIN_ARG_INFO(arginfo_iterator_it___construct, 0)
3111:     ZEND_ARG_OBJ_INFO(0, iterator, Traversable, 0)
3112: ZEND_END_ARG_INFO();
3113:
3114: static const zend_function_entry spl_func_IteratorIterator[] = {
3115:     SPL_ME(IteratorIterator, __construct, arginfo_iterator_it___construct, ZEND_ACC_PUBLIC)
3116:     SPL_ME(dual_it, rewind, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3117:     SPL_ME(dual_it, valid, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3118:     SPL_ME(dual_it, key, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3119:     SPL_ME(dual_it, current, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3120:     SPL_ME(dual_it, next, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3121:     SPL_ME(dual_it, getInnerIterator, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3122:     PHP_FE_END
3123: };
3124:
3125: /* {{{ proto void NoRewindIterator::__construct(Iterator it)
3126:  * Create an iterator from another iterator */
3127: SPL_METHOD(NoRewindIterator, __construct)
3128: {
3129:     spl_dual_it_construct (INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ce_NoRewindIterator, zend_ce_iterator, DIT_NoRewindIterator);
3130: } /* }}} */
3131:
3132: /* {{{ proto void NoRewindIterator::rewind()
3133:  * Prevent a call to inner iterators rewind() */
3134: SPL_METHOD(NoRewindIterator, rewind)
3135: {
3136:     if (zend_parse_parameters_none() == FAILURE) {
3137:         return;
3138:     }
3139:
3140:     /* nothing to do */
3141: } /* }}} */
3142:
3143: /* {{{ proto bool NoRewindIterator::valid()
3144:  * Return inner iterator's valid() */
3145: SPL_METHOD(NoRewindIterator, valid)
3146: {
3147:     spl_dual_it_obj_t *intern;
3148:
3149:     if (zend_parse_parameters_none() == FAILURE) {
3150:         return;
3151:     }
3152:
3153:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3154:
3155:     if (intern->inner.iterator->funcs->get_current_key(intern->inner.iterator, return_value);
3156:         ) else {
3157:             RETURN_NULL();
3158:         }
3159: } /* }}} */
3160:
3161: /* {{{ proto mixed NoRewindIterator::key()
3162:  * Return inner iterators key() */
3163: SPL_METHOD(NoRewindIterator, key)
3164: {
3165:     spl_dual_it_obj_t *intern;
3166:
3167:     if (zend_parse_parameters_none() == FAILURE) {
3168:         return;
3169:     }
3170:
3171:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3172:
3173:     if (intern->inner.iterator->funcs->get_current_key(intern->inner.iterator, return_value);
3174:         ) else {
3175:             RETURN_NULL();
3176:         }
3177: } /* }}} */
3178:
3179: /* {{{ proto mixed NoRewindIterator::next()
3180:  * Return inner iterators next() */
3181: SPL_METHOD(NoRewindIterator, next)
3182: {
3183:     spl_dual_it_obj_t *intern;
3184:
3185:     if (zend_parse_parameters_none() == FAILURE) {
3186:         return;
3187:     }
3188:
3189:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3190:
3191:     if (intern->inner.iterator->funcs->move_forward(intern->inner.iterator);
3192:         ) else {
3193:             RETURN_NULL();
3194:         }
3195: } /* }}} */
3196:
3197: static const zend_function_entry spl_func_NoRewindIterator[] = {
3198:     SPL_ME(NoRewindIterator, __construct, arginfo_no_rewind_it___construct, ZEND_ACC_PUBLIC)
3199:     SPL_ME(NoRewindIterator, rewind, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3200:     SPL_ME(NoRewindIterator, valid, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3201:     SPL_ME(NoRewindIterator, key, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3202:     SPL_ME(NoRewindIterator, next, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3203:     PHP_FE_END
3204: };
3205:
3206: /* {{{ proto void InfiniteIterator::__construct(Iterator it)
3207:  * Create an iterator from another iterator */
3208: SPL_METHOD(InfiniteIterator, __construct)
3209: {
3210:     spl_dual_it_construct (INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ce_InfiniteIterator, zend_ce_iterator, DIT_InfiniteIterator);
3211: } /* }}} */
3212:
3213: /* {{{ proto void InfiniteIterator::next()
3214:  * Prevent a call to inner iterators rewind() (internally the current data will be fetched if valid) */
3215: SPL_METHOD(InfiniteIterator, next)
3216: {
3217:     spl_dual_it_obj_t *intern;
3218:
3219:     if (zend_parse_parameters_none() == FAILURE) {
3220:         return;
3221:     }
3222:
3223:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3224:
3225:     if (zend_parse_parameters_none() == FAILURE) {
3226:         return;
3227:     }
3228:
3229:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3230:
3231:     if (zend_parse_parameters_none() == FAILURE) {
3232:         return;
3233:     }
3234:
3235:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3236:
3237:     if (zend_parse_parameters_none() == FAILURE) {
3238:         return;
3239:     }
3240:
3241:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3242:
3243:     if (zend_parse_parameters_none() == FAILURE) {
3244:         return;
3245:     }
3246:
3247:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3248:
3249:     if (zend_parse_parameters_none() == FAILURE) {
3250:         return;
3251:     }
3252:
3253:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3254:
3255:     if (zend_parse_parameters_none() == FAILURE) {
3256:         return;
3257:     }
3258:
3259:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3260:
3261:     if (zend_parse_parameters_none() == FAILURE) {
3262:         return;
3263:     }
3264:
3265:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3266:
3267:     if (zend_parse_parameters_none() == FAILURE) {
3268:         return;
3269:     }
3270:
3271:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3272:
3273:     if (zend_parse_parameters_none() == FAILURE) {
3274:         return;
3275:     }
3276:
3277:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3278:
3279:     if (zend_parse_parameters_none() == FAILURE) {
3280:         return;
3281:     }
3282:
3283:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3284:
3285:     if (zend_parse_parameters_none() == FAILURE) {
3286:         return;
3287:     }
3288:
3289:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3290:
3291:     if (zend_parse_parameters_none() == FAILURE) {
3292:         return;
3293:     }
3294:
3295:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3296:
3297:     if (zend_parse_parameters_none() == FAILURE) {
3298:         return;
3299:     }
3300:
3301:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3302:
3303:     if (zend_parse_parameters_none() == FAILURE) {
3304:         return;
3305:     }
3306:
3307:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3308:
3309:     if (zend_parse_parameters_none() == FAILURE) {
3310:         return;
3311:     }
3312:
3313:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3314:
3315:     if (zend_parse_parameters_none() == FAILURE) {
3316:         return;
3317:     }
3318:
3319:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3320:
3321:     if (zend_parse_parameters_none() == FAILURE) {
3322:         return;
3323:     }
3324:
3325:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3326:
3327:     if (zend_parse_parameters_none() == FAILURE) {
3328:         return;
3329:     }
3330:
3331:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3332:
3333:     if (zend_parse_parameters_none() == FAILURE) {
3334:         return;
3335:     }
3336:
3337:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3338:
3339:     if (zend_parse_parameters_none() == FAILURE) {
3340:         return;
3341:     }
3342:
3343:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3344:
3345:     if (zend_parse_parameters_none() == FAILURE) {
3346:         return;
3347:     }
3348:
3349:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3350:
3351:     if (zend_parse_parameters_none() == FAILURE) {
3352:         return;
3353:     }
3354:
3355:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3356:
3357:     if (zend_parse_parameters_none() == FAILURE) {
3358:         return;
3359:     }
3360:
3361:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3362:
3363:     if (zend_parse_parameters_none() == FAILURE) {
3364:         return;
3365:     }
3366:
3367:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3368:
3369:     if (zend_parse_parameters_none() == FAILURE) {
3370:         return;
3371:     }
3372:
3373:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3374:
3375:     if (zend_parse_parameters_none() == FAILURE) {
3376:         return;
3377:     }
3378:
3379:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3380:
3381:     if (zend_parse_parameters_none() == FAILURE) {
3382:         return;
3383:     }
3384:
3385:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3386:
3387:     if (zend_parse_parameters_none() == FAILURE) {
3388:         return;
3389:     }
3390:
3391:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3392:
3393:     if (zend_parse_parameters_none() == FAILURE) {
3394:         return;
3395:     }
3396:
3397:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3398:
3399:     if (zend_parse_parameters_none() == FAILURE) {
3400:         return;
3401:     }
3402:
3403:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3404:
3405:     if (zend_parse_parameters_none() == FAILURE) {
3406:         return;
3407:     }
3408:
3409:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3410:
3411:     if (zend_parse_parameters_none() == FAILURE) {
3412:         return;
3413:     }
3414:
3415:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3416:
3417:     if (zend_parse_parameters_none() == FAILURE) {
3418:         return;
3419:     }
3420:
3421:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3422:
3423:     if (zend_parse_parameters_none() == FAILURE) {
3424:         return;
3425:     }
3426:
3427:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3428:
3429:     if (zend_parse_parameters_none() == FAILURE) {
3430:         return;
3431:     }
3432:
3433:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3434:
3435:     if (zend_parse_parameters_none() == FAILURE) {
3436:         return;
3437:     }
3438:
3439:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3440:
3441:     if (zend_parse_parameters_none() == FAILURE) {
3442:         return;
3443:     }
3444:
3445:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3446:
3447:     if (zend_parse_parameters_none() == FAILURE) {
3448:         return;
3449:     }
3450:
3451:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3452:
3453:     if (zend_parse_parameters_none() == FAILURE) {
3454:         return;
3455:     }
3456:
3457:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3458:
3459:     if (zend_parse_parameters_none() == FAILURE) {
3460:         return;
3461:     }
3462:
3463:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3464:
3465:     if (zend_parse_parameters_none() == FAILURE) {
3466:         return;
3467:     }
3468:
3469:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3470:
3471:     if (zend_parse_parameters_none() == FAILURE) {
3472:         return;
3473:     }
3474:
3475:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3476:
3477:     if (zend_parse_parameters_none() == FAILURE) {
3478:         return;
3479:     }
3480:
3481:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3482:
3483:     if (zend_parse_parameters_none() == FAILURE) {
3484:         return;
3485:     }
3486:
3487:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3488:
3489:     if (zend_parse_parameters_none() == FAILURE) {
3490:         return;
3491:     }
3492:
3493:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3494:
3495:     if (zend_parse_parameters_none() == FAILURE) {
3496:         return;
3497:     }
3498:
3499:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3500:
3501:     if (zend_parse_parameters_none() == FAILURE) {
3502:         return;
3503:     }
3504:
3505:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3506:
3507:     if (zend_parse_parameters_none() == FAILURE) {
3508:         return;
3509:     }
3510:
3511:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3512:
3513:     if (zend_parse_parameters_none() == FAILURE) {
3514:         return;
3515:     }
3516:
3517:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3518:
3519:     if (zend_parse_parameters_none() == FAILURE) {
3520:         return;
3521:     }
3522:
3523:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3524:
3525:     if (zend_parse_parameters_none() == FAILURE) {
3526:         return;
3527:     }
3528:
3529:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3530:
3531:     if (zend_parse_parameters_none() == FAILURE) {
3532:         return;
3533:     }
3534:
3535:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3536:
3537:     if (zend_parse_parameters_none() == FAILURE) {
3538:         return;
3539:     }
3540:
3541:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3542:
3543:     if (zend_parse_parameters_none() == FAILURE) {
3544:         return;
3545:     }
3546:
3547:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3548:
3549:     if (zend_parse_parameters_none() == FAILURE) {
3550:         return;
3551:     }
3552:
3553:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3554:
3555:     if (zend_parse_parameters_none() == FAILURE) {
3556:         return;
3557:     }
3558:
3559:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3560:
3561:     if (zend_parse_parameters_none() == FAILURE) {
3562:         return;
3563:     }
3564:
3565:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3566:
3567:     if (zend_parse_parameters_none() == FAILURE) {
3568:         return;
3569:     }
3570:
3571:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3572:
3573:     if (zend_parse_parameters_none() == FAILURE) {
3574:         return;
3575:     }
3576:
3577:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3578:
3579:     if (zend_parse_parameters_none() == FAILURE) {
3580:         return;
3581:     }
3582:
3583:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3584:
3585:     if (zend_parse_parameters_none() == FAILURE) {
3586:         return;
3587:     }
3588:
3589:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3590:
3591:     if (zend_parse_parameters_none() == FAILURE) {
3592:         return;
3593:     }
3594:
3595:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3596:
3597:     if (zend_parse_parameters_none() == FAILURE) {
3598:         return;
3599:     }
3600:
3601:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3602:
3603:     if (zend_parse_parameters_none() == FAILURE) {
3604:         return;
3605:     }
3606:
3607:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3608:
3609:     if (zend_parse_parameters_none() == FAILURE) {
3610:         return;
3611:     }
3612:
3613:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3614:
3615:     if (zend_parse_parameters_none() == FAILURE) {
3616:         return;
3617:     }
3618:
3619:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3620:
3621:     if (zend_parse_parameters_none() == FAILURE) {
3622:         return;
3623:     }
3624:
3625:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3626:
3627:     if (zend_parse_parameters_none() == FAILURE) {
3628:         return;
3629:     }
3630:
3631:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3632:
3633:     if (zend_parse_parameters_none() == FAILURE) {
3634:         return;
3635:     }
3636:
3637:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3638:
3639:     if (zend_parse_parameters_none() == FAILURE) {
3640:         return;
3641:     }
3642:
3643:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3644:
3645:     if (zend_parse_parameters_none() == FAILURE) {
3646:         return;
3647:     }
3648:
3649:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3650:
3651:     if (zend_parse_parameters_none() == FAILURE) {
3652:         return;
3653:     }
3654:
3655:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3656:
3657:     if (zend_parse_parameters_none() == FAILURE) {
3658:         return;
3659:     }
3660:
3661:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3662:
3663:     if (zend_parse_parameters_none() == FAILURE) {
3664:         return;
3665:     }
3666:
3667:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3668:
3669:     if (zend_parse_parameters_none() == FAILURE) {
3670:         return;
3671:     }
3672:
3673:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3674:
3675:     if (zend_parse_parameters_none() == FAILURE) {
3676:         return;
3677:     }
3678:
3679:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3680:
3681:     if (zend_parse_parameters_none() == FAILURE) {
3682:         return;
3683:     }
3684:
3685:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3686:
3687:     if (zend_parse_parameters_none() == FAILURE) {
3688:         return;
3689:     }
3690:
3691:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3692:
3693:     if (zend_parse_parameters_none() == FAILURE) {
3694:         return;
3695:     }
3696:
3697:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3698:
3699:     if (zend_parse_parameters_none() == FAILURE) {
3700:         return;
3701:     }
3702:
3703:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3704:
3705:     if (zend_parse_parameters_none() == FAILURE) {
3706:         return;
3707:     }
3708:
3709:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3710:
3711:     if (zend_parse_parameters_none() == FAILURE) {
3712:         return;
3713:     }
3714:
3715:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3716:
3717:     if (zend_parse_parameters_none() == FAILURE) {
3718:         return;
3719:     }
3720:
3721:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3722:
3723:     if (zend_parse_parameters_none() == FAILURE) {
3724:         return;
3725:     }
3726:
3727:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3728:
3729:     if (zend_parse_parameters_none() == FAILURE) {
3730:         return;
3731:     }
3732:
3733:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3734:
3735:     if (zend_parse_parameters_none() == FAILURE) {
3736:         return;
3737:     }
3738:
3739:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3740:
3741:     if (zend_parse_parameters_none() == FAILURE) {
3742:         return;
3743:     }
3744:
3745:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3746:
3747:     if (zend_parse_parameters_none() == FAILURE) {
3748:         return;
3749:     }
3750:
3751:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3752:
3753:     if (zend_parse_parameters_none() == FAILURE) {
3754:         return;
3755:     }
3756:
3757:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3758:
3759:     if (zend_parse_parameters_none() == FAILURE) {
3760:         return;
3761:     }
3762:
3763:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3764:
3765:     if (zend_parse_parameters_none() == FAILURE) {
3766:         return;
3767:     }
3768:
3769:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3770:
3771:     if (zend_parse_parameters_none() == FAILURE) {
3772:         return;
3773:     }
3774:
3775:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3776:
3777:     if (zend_parse_parameters_none() == FAILURE) {
3778:         return;
3779:     }
3780:
3781:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3782:
3783:     if (zend_parse_parameters_none() == FAILURE) {
3784:         return;
3785:     }
3786:
3787:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3788:
3789:     if (zend_parse_parameters_none() == FAILURE) {
3790:         return;
3791:     }
3792:
3793:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3794:
3795:     if (zend_parse_parameters_none() == FAILURE) {
3796:         return;
3797:     }
3798:
3799:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3800:
3801:     if (zend_parse_parameters_none() == FAILURE) {
3802:         return;
3803:     }
3804:
3805:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3806:
3807:     if (zend_parse_parameters_none() == FAILURE) {
3808:         return;
3809:     }
3810:
3811:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3812:
3813:     if (zend_parse_parameters_none() == FAILURE) {
3814:         return;
3815:     }
3816:
3817:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3818:
3819:     if (zend_parse_parameters_none() == FAILURE) {
3820:         return;
3821:     }
3822:
3823:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3824:
3825:     if (zend_parse_parameters_none() == FAILURE) {
3826:         return;
3827:     }
3828:
3829:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3830:
3831:     if (zend_parse_parameters_none() == FAILURE) {
3832:         return;
3833:     }
3834:
3835:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3836:
3837:     if (zend_parse_parameters_none() == FAILURE) {
3838:         return;
3839:     }
3840:
3841:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3842:
3843:     if (zend_parse_parameters_none() == FAILURE) {
3844:         return;
3845:     }
3846:
3847:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3848:
3849:     if (zend_parse_parameters_none() == FAILURE) {
3850:         return;
3851:     }
3852:
3853:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3854:
3855:     if (zend_parse_parameters_none() == FAILURE) {
3856:         return;
3857:     }
3858:
3859:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3860:
3861:     if (zend_parse_parameters_none() == FAILURE) {
3862:         return;
3863:     }
3864:
3865:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3866:
3867:     if (zend_parse_parameters_none() == FAILURE) {
3868:         return;
3869:     }
3870:
3871:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3872:
3873:     if (zend_parse_parameters_none() == FAILURE) {
3874:         return;
3875:     }
3876:
3877:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3878:
3879:     if (zend_parse_parameters_none() == FAILURE) {
3880:         return;
3881:     }
3882:
3883:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3884:
3885:     if (zend_parse_parameters_none() == FAILURE) {
3886:         return;
3887:     }
3888:
3889:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3890:
3891:     if (zend_parse_parameters_none() == FAILURE) {
3892:         return;
3893:     }
3894:
3895:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3896:
3897:     if (zend_parse_parameters_none() == FAILURE) {
3898:         return;
3899:     }
3900:
3901:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3902:
3903:     if (zend_parse_parameters_none() == FAILURE) {
3904:         return;
3905:     }
3906:
3907:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3908:
3909:     if (zend_parse_parameters_none() == FAILURE) {
3910:         return;
3911:     }
3912:
3913:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3914:
3915:     if (zend_parse_parameters_none() == FAILURE) {
3916:         return;
3917:     }
3918:
3919:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3920:
3921:     if (zend_parse_parameters_none() == FAILURE) {
3922:         return;
3923:     }
3924:
3925:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3926:
3927:     if (zend_parse_parameters_none() == FAILURE) {
3928:         return;
3929:     }
3930:
3931:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3932:
3933:     if (zend_parse_parameters_none() == FAILURE) {
3934:         return;
3935:     }
3936:
3937:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3938:
3939:     if (zend_parse_parameters_none() == FAILURE) {
3940:         return;
3941:     }
3942:
3943:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3944:
3945:     if (zend_parse_parameters_none() == FAILURE) {
3946:         return;
3947:     }
3948:
3949:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3950:
3951:     if (zend_parse_parameters_none() == FAILURE) {
3952:         return;
3953:     }
3954:
3955:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3956:
3957:     if (zend_parse_parameters_none() == FAILURE) {
3958:         return;
3959:     }
3960:
3961:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3962:
3963:     if (zend_parse_parameters_none() == FAILURE) {
3964:         return;
3965:     }
3966:
3967:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3968:
3969:     if (zend_parse_parameters_none() == FAILURE) {
3970:         return;
3971:     }
3972:
3973:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3974:
3975:     if (zend_parse_parameters_none() == FAILURE) {
3976:         return;
3977:     }
3978:
3979:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3980:
3981:     if (zend_parse_parameters_none() == FAILURE) {
3982:         return;
3983:     }
3984:
3985:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3986:
3987:     if (zend_parse_parameters_none() == FAILURE) {
3988:         return;
3989:     }
3990:
3991:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3992:
3993:     if (zend_parse_parameters_none() == FAILURE) {
3994:         return;
3995:     }
3996:
3997:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3998:
3999:     if (zend_parse_parameters_none() == FAILURE) {
4000:         return;
4001:     }
4002:
4003:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis
```



```
3738: REGISTER_SPL_STD_CLASS_EX(IteratorIterator, spl_dual_it_new, spl_funcns_IteratorIterator);
3739: REGISTER_SPL_ITERATOR(IteratorIterator);
3740: REGISTER_SPL_ITERATOR(IteratorIterator);
3741: REGISTER_SPL_IMPLMENTS(IteratorIterator, OuterIterator);
3742:
3743: REGISTER_SPL_SUB_CLASS_EX(FilterIterator, IteratorIterator, spl_dual_it_new, spl_funcns_FilterIterator);
3744: spl_on_FilterIterator->on_flags |= ZEND_ACC_EXPLICIT_ABSTRACT_CLASS;
3745:
3746: REGISTER_SPL_SUB_CLASS_EX(RecursiveFilterIterator, FilterIterator, spl_dual_it_new, spl_funcns_RecursiveFilterIterator);
3747: REGISTER_SPL_IMPLMENTS(RecursiveFilterIterator, RecursiveIterator);
3748:
3749: REGISTER_SPL_SUB_CLASS_EX(CallbackFilterIterator, FilterIterator, spl_dual_it_new, spl_funcns_CallbackFilterIterator);
3750:
3751: REGISTER_SPL_SUB_CLASS_EX(RecursiveCallbackFilterIterator, CallbackFilterIterator, spl_dual_it_new, spl_funcns_RecursiveCallbackFilterIterator);
3752: REGISTER_SPL_IMPLMENTS(RecursiveCallbackFilterIterator, RecursiveIterator);
3753:
3754:
3755: REGISTER_SPL_SUB_CLASS_EX(ParentIterator, RecursiveFilterIterator, spl_dual_it_new, spl_funcns_ParentIterator);
3756:
3757: REGISTER_SPL_INTERFACE(SeekableIterator);
3758: REGISTER_SPL_ITERATOR(SeekableIterator);
3759:
3760: REGISTER_SPL_SUB_CLASS_EX(LimitIterator, IteratorIterator, spl_dual_it_new, spl_funcns_LimitIterator);
3761:
3762: REGISTER_SPL_SUB_CLASS_EX(CachingIterator, IteratorIterator, spl_dual_it_new, spl_funcns_CachingIterator);
3763: REGISTER_SPL_IMPLMENTS(CachingIterator, ArrayAccess);
3764: REGISTER_SPL_IMPLMENTS(CachingIterator, Countable);
3765:
3766: REGISTER_SPL_CLASS_CONST_LONG(CachingIterator, "CALL_TOSTRING", CIT_CALL_TOSTRING);
3767: REGISTER_SPL_CLASS_CONST_LONG(CachingIterator, "CATCH_GET_CHILD", CIT_CATCH_GET_CHILD);
3768: REGISTER_SPL_CLASS_CONST_LONG(CachingIterator, "TOSTRING_USE_KEY", CIT_TOSTRING_USE_KEY);
3769: REGISTER_SPL_CLASS_CONST_LONG(CachingIterator, "TOSTRING_USE_CURRENT", CIT_TOSTRING_USE_CURRENT);
3770: REGISTER_SPL_CLASS_CONST_LONG(CachingIterator, "TOSTRING_USE_INNER", CIT_TOSTRING_USE_INNER);
3771: REGISTER_SPL_CLASS_CONST_LONG(CachingIterator, "FULL_CACHE", CIT_FULL_CACHE);
3772:
3773: REGISTER_SPL_SUB_CLASS_EX(RecursiveCachingIterator, CachingIterator, spl_dual_it_new, spl_funcns_RecursiveCachingIterator);
3774: REGISTER_SPL_IMPLMENTS(RecursiveCachingIterator, RecursiveIterator);
3775:
3776: REGISTER_SPL_SUB_CLASS_EX(NoRewindIterator, IteratorIterator, spl_dual_it_new, spl_funcns_NoRewindIterator);
3777:
3778: REGISTER_SPL_SUB_CLASS_EX(AppendIterator, IteratorIterator, spl_dual_it_new, spl_funcns_AppendIterator);
3779:
3780: REGISTER_SPL_IMPLMENTS(RecursiveIteratorIterator, OuterIterator);
3781:
3782: REGISTER_SPL_SUB_CLASS_EX(InfinitelIterator, IteratorIterator, spl_dual_it_new, spl_funcns_InfinitelIterator);
3783:
3784: #if HAVE_PCRE || HAVE_REGEX_PCRE
3785: REGISTER_SPL_SUB_CLASS_EX(RegexIterator, FilterIterator, spl_dual_it_new, spl_funcns_RegexIterator);
3786: REGISTER_SPL_CLASS_CONST_LONG(RegexIterator, "USE_KEY", REGIT_USE_KEY);
3787: REGISTER_SPL_CLASS_CONST_LONG(RegexIterator, "INVERT_MATCH", REGIT_INVERTED);
3788: REGISTER_SPL_CLASS_CONST_LONG(RegexIterator, "MATCH", REGIT_MATCH_MATCH);
3789: REGISTER_SPL_CLASS_CONST_LONG(RegexIterator, "GET_MATCH", REGIT_MATCH_GET_MATCH);
3790: REGISTER_SPL_CLASS_CONST_LONG(RegexIterator, "ALL_MATCHES", REGIT_MATCH_ALL_MATCHES);
3791: REGISTER_SPL_CLASS_CONST_LONG(RegexIterator, "SPLIT", REGIT_MATCH_SPLIT);
3792: REGISTER_SPL_CLASS_CONST_LONG(RegexIterator, "REPLACE", REGIT_MATCH_REPLACE);
3793: REGISTER_SPL_CLASS_CONST_LONG(RegexIterator, "PROPERTY", REGIT_MATCH_PROPERTY);
3794: REGISTER_SPL_IMPLMENTS(RecursiveRegexIterator, RecursiveIterator);
3795:
3796: spl_on_RegexIterator = NULL;
3797: spl_on_RecursiveRegexIterator = NULL;
3798: #endif
3799:
3800: REGISTER_SPL_STD_CLASS_EX(EmptyIterator, NULL, spl_funcns_EmptyIterator);
3801: REGISTER_SPL_ITERATOR(EmptyIterator);
3802:
3803: REGISTER_SPL_SUB_CLASS_EX(RecursiveTreeIterator, RecursiveIteratorIterator, spl_RecursiveTreeIterator_new, spl_funcns_RecursiveTreeIterator);
3804: REGISTER_SPL_CLASS_CONST_LONG(RecursiveTreeIterator, "BYPASS_CURRENT", RTIT_BYPASS_CURRENT);
3805: REGISTER_SPL_CLASS_CONST_LONG(RecursiveTreeIterator, "BYPASS_KEY", RTIT_BYPASS_KEY);
3806: REGISTER_SPL_CLASS_CONST_LONG(RecursiveTreeIterator, "PREFIX_LEFT", 0);
3807: REGISTER_SPL_CLASS_CONST_LONG(RecursiveTreeIterator, "PREFIX_MID_HAS_NEXT", 1);
3808: REGISTER_SPL_CLASS_CONST_LONG(RecursiveTreeIterator, "PREFIX_MID_LAST", 2);
3809: REGISTER_SPL_CLASS_CONST_LONG(RecursiveTreeIterator, "PREFIX_END_HAS_NEXT", 3);
3810: REGISTER_SPL_CLASS_CONST_LONG(RecursiveTreeIterator, "PREFIX_END_LAST", 4);
3811: REGISTER_SPL_CLASS_CONST_LONG(RecursiveTreeIterator, "PREFIX_RIGHT", 5);
3812:
3813: return SUCCESS;
3814: }
3815: /* }}} */
3816:
3817: /*
3818:  * Local variables:
3819:  * tab-width: 4
3820:  * c-basic-offset: 4
3821:  * End:
3822:  * vim600: fdm=marker
3823:  * vim: noet sw=4 ts=4
3824:  */
```

```
1: /*
2:  *-----
3:  * | PHP Version 7 |
4:  *-----
5:  * | Copyright (c) 1997-2018 The PHP Group |
6:  *-----
7:  * | This source file is subject to version 3.01 of the PHP license, |
8:  * | that is bundled with this package in the file LICENSE, and is |
9:  * | available through the world-wide-web at the following url: |
10:  * | http://www.php.net/license/3.01.txt |
11:  * | If you did not receive a copy of the PHP license and are unable to |
12:  * | obtain it through the world-wide-web, please send a note to |
13:  * | license@php.net so we can mail you a copy immediately. |
14:  *-----
15:  * | Authors: Etienne Kneuss <colder@php.net> |
16:  *-----
17:  */
18:
19: /* $Id$ */
20:
21: #ifndef SPL_DLIST_H
22: #define SPL_DLIST_H
23:
24: #include "php.h"
25: #include "php_spl.h"
26:
27: extern PHPAPI zend_class_entry *spl_ce_SplDoublyLinkedList;
28: extern PHPAPI zend_class_entry *spl_ce_SplQueue;
29: extern PHPAPI zend_class_entry *spl_ce_SplStack;
30:
31: PHP_MINIT_FUNCTION(spl_dlist);
32:
33: #endif /* SPL_DLIST_H */
34:
35: /*
36:  * Local Variables:
37:  * n-basis-offset: 4
38:  * tab-width: 4
39:  * End:
40:  * vim600: fdm=marker
41:  * vim: noet sw=4 ts=4
42:  */
```

```
1: /*
2:  * -----
3:  * | PHP Version ? |
4:  * -----
5:  * | Copyright (c) 1997-2018 The PHP Group |
6:  * -----
7:  * | This source file is subject to version 3.01 of the PHP license, |
8:  * | that is bundled with this package in the file LICENSE, and is |
9:  * | available through the world-wide-web at the following url: |
10:  * | http://www.php.net/license/3.01.txt |
11:  * | If you did not receive a copy of the PHP license and are unable to |
12:  * | obtain it through the world-wide-web, please send a note to |
13:  * | license@php.net so we can mail you a copy immediately. |
14:  * -----
15:  * | Authors: Marcus Boerger <helly@php.net> |
16:  * -----
17:  */
18:
19: /* $Id$ */
20:
21: #ifndef PHP_FUNCTIONS_H
22: #define PHP_FUNCTIONS_H
23:
24: #include "php.h"
25:
26: typedef zend_object* (*create_object_func_t)(zend_class_entry *class_type);
27:
28: #define REGISTER_SPL_STO_CLASS(class_name, obj_ctor) \
29:     spl_register_std_class(spl_ce_ ## class_name, # class_name, obj_ctor, NULL);
30:
31: #define REGISTER_SPL_STO_CLASS_EX(class_name, obj_ctor, funcns) \
32:     spl_register_std_class(spl_ce_ ## class_name, # class_name, obj_ctor, funcns);
33:
34: #define REGISTER_SPL_SUB_CLASS_EX(class_name, parent_class_name, obj_ctor, funcns) \
35:     spl_register_sub_class(spl_ce_ ## class_name, spl_ce_ ## parent_class_name, # class_name, obj_ctor, funcns);
36:
37: #define REGISTER_SPL_INTERFACE(class_name) \
38:     spl_register_interface(spl_ce_ ## class_name, # class_name, spl_funcns_ ## class_name);
39:
40: #define REGISTER_SPL_IMPLEMENTOR(class_name, interface_name) \
41:     zend_class_implements(spl_ce_ ## class_name, 1, spl_ce_ ## interface_name);
42:
43: #define REGISTER_SPL_ITERATOR(class_name) \
44:     zend_class_implements(spl_ce_ ## class_name, 1, zend_ce_iterator);
45:
46: #define REGISTER_SPL_PROPERTY(class_name, prop_name, prop_flags) \
47:     spl_register_property(spl_ce_ ## class_name, prop_name, sizeof(prop_name)-1, prop_flags);
48:
49: #define REGISTER_SPL_CLASS_CONST_LONG(class_name, const_name, value) \
50:     zend_declare_class_constant_long(spl_ce_ ## class_name, const_name, sizeof(const_name)-1, (zend_long)value);
51:
52: void spl_register_std_class(zend_class_entry ** pcew, char * class_name, create_object_func_t ctor, const zend_function_entry * function_list);
53: void spl_register_sub_class(zend_class_entry ** pcew, zend_class_entry * parent_ce, char * class_name, create_object_func_t ctor, const zend_function_
ntry * function_list);
54: void spl_register_interface(zend_class_entry ** pcew, char * class_name, const zend_function_entry * functions);
55:
56: void spl_register_property(zend_class_entry * class_entry, char *prop_name, int prop_name_len, int prop_flags);
57:
58: /* sub: whether to allow subclasses/interfaces
59:    allow = 0: allow all classes and interfaces
60:    allow > 0: allow all that match and mask ce_flags
61:    allow < 0: disallow all that match and mask ce_flags
62: */
63: void spl_add_class_name(zval * list, zend_class_entry * pce, int allow, int ce_flags);
64: void spl_add_interfaces(zval * list, zend_class_entry * pce, int allow, int ce_flags);
65: void spl_add_traits(zval * list, zend_class_entry * pce, int allow, int ce_flags);
66: int spl_add_classes(zend_class_entry *pce, zval *list, int sub, int allow, int ce_flags);
67:
68: /* caller must free(return) */
69: zend_string *spl_get_private_prop_name(zend_class_entry *ce, char *prop_name, int prop_len);
70:
71: #define SPL_ME(class_name, function_name, arg_info, flags) \
72:     PHP_ME(spl, ## class_name, function_name, arg_info, flags)
73:
74: #define SPL_ABSTRACT_ME(class_name, function_name, arg_info) \
75:     ZEND_ABSTRACT_ME(spl, ## class_name, function_name, arg_info)
76:
77: #define SPL_METHOD(class_name, function_name) \
78:     PHP_METHOD(spl, ## class_name, function_name)
79:
80: #define SPL_MA(class_name, function_name, alias_class, alias_function, arg_info, flags) \
81:     PHP_ALIAS(spl, ## alias_class, function_name, alias_function, arg_info, flags)
82: #endif /* PHP_FUNCTIONS_H */
83:
84: /*
85:  * Local Variables:
86:  * c-basic-offset: 4
87:  * tab-width: 4
88:  * End:
89:  * vim600: fdm=marker
90:  * vim: noet sw=4 ts=4
91:  */
```

[illegible]

```

377:  if (aif->func_ptr &&
378:      UNEXPECTED(aif->func_ptr->common.fn_flags & ZEND_ACC_CALL_VIA_TRAMPOLINE)) {
379:      zend_string_release(aif->func_ptr->common.function_name);
380:      zend_free_trampoline(aif->func_ptr);
381:  }
382:  if (!IS_UNDEF(aif->closure)) {
383:      zval_ptr_dtor(aif->closure);
384:  }
385:  efree(aif);
386:  }
387:
388:  /* {{{ proto void spl_autoload_call(string class_name)
389:   * Try all registered autoload function to load the requested class */
390:  PHP_FUNCTION(spl_autoload_call)
391:  {
392:      zval *class_name, *retval;
393:      zend_string *lc_name, *func_name;
394:      autoload_func_info *aif;
395:  }
396:  if (zend_parse_parameters(ZEND_NUM_ARGS() | "s", &class_name) == FAILURE || Z_TYPE_P(class_name) != IS_STRING) {
397:      return;
398:  }
399:  }
400:  if (SPL_G(autoload_functions)) {
401:      hashPosition pos;
402:      zend_ulong num_idx;
403:      zend_function *func;
404:      zend_fcall_info fci;
405:      zend_fcall_info_cache fci_cache;
406:      zend_class_entry *called_scope = zend_get_called_scope(execute_data);
407:      int i_automload_running = SPL_G(autoload_running);
408:  }
409:  SPL_G(autoload_running) = 1;
410:  lc_name = zend_string_tolower(Z_STR_P(class_name));
411:  }
412:  fci.size = sizeof(fci);
413:  fci.retval = &retval;
414:  fci.param_count = 1;
415:  fci.params = class_name;
416:  fci.no_separation = 1;
417:  }
418:  ZVAL_UNDEF(&fci.function_name); /* Unused */
419:  }
420:  zend_hash_internal_pointer_reset_ex(SPL_G(autoload_functions), &pos);
421:  while (zend_hash_get_current_key_ex(SPL_G(autoload_functions), &func_name, &num_idx, &pos) == HASH_KEY_IS_STRING) {
422:      aif = zend_hash_get_current_data_ptr_ex(SPL_G(autoload_functions), &pos);
423:      func = aif->func_ptr;
424:      if (UNEXPECTED(func->common.fn_flags & ZEND_ACC_CALL_VIA_TRAMPOLINE)) {
425:          func = &autoload_trampoline;
426:          memcpy(func, aif->func_ptr, sizeof(send_up_array));
427:          zend_string_addr(func->op_array.function_name);
428:      }
429:      ZVAL_UNDEF(&ret_val);
430:      fci.func.handler = func;
431:      if (!IS_UNDEF(aif->obj)) {
432:          fci.object = &obj;
433:      } else {
434:          fci.object = NULL;
435:      }
436:      fci.calling_scope = aif->obj;
437:      if (aif->obj &&
438:          (called_scope ||
439:           !instanceof_function(called_scope, aif->obj))) {
440:          fci.called_scope = aif->obj;
441:      } else {
442:          fci.called_scope = called_scope;
443:      }
444:      fci.object = &obj;
445:      fci.called_scope = &obj;
446:      }
447:  }
448:  zend_call_function(&fci, &fci_cache);
449:  zval_ptr_dtor(&ret_val);
450:  }
451:  if (EG(exception)) {
452:      break;
453:  }
454:  if (pos + 1 == SPL_G(autoload_functions)->nNumOfElements) {
455:      zend_hash_exists(&EG(class_table), lc_name);
456:      break;
457:  }
458:  zend_hash_move_forward_ex(SPL_G(autoload_functions), &pos);
459:  }
460:  zend_string_release(lc_name);
461:  SPL_G(autoload_running) = 1;
462:  } else {
463:      /* do not use or overwrite EG(autoload_func) here */
464:      zend_call_method_with_1_params(NULL, NULL, NULL, "spl_autoload", NULL, class_name);
465:  }
466:  }
467:  /* }}} */
468:  }
469:  zend_time HT_MOVE_TAIL_TO_HEAD(ht)
470:  {
471:      Bucket tmp = (ht->data[(ht->nNumUsed-1)]);
472:      memmove(&ht->data + 1, &tmp, &ht->nNumUsed - 1);
473:      tmp = ht->data[0];
474:      ht->data[0] = tmp;
475:      zend_hash_rehash(ht);
476:      } while (0)
477:  }
478:  /* {{{ proto bool spl_autoload_register([mixed autoload_function [, bool throw [, bool prepend]]])
479:   * Register given function as __autoload() implementation */
480:  PHP_FUNCTION(spl_autoload_register)
481:  {
482:      zend_string *func_name;
483:      char *error = NULL;
484:      zend_string *lc_name;
485:      zval *callable = NULL;
486:      zend_bool do_throw = 1;
487:      zend_bool prepend = 0;
488:      zend_function *aif_func_ptr;
489:      autoload_func_info aif;
490:      zend_object *obj_ptr;
491:      zend_fcall_info_cache fci;
492:  }
493:  if (zend_parse_parameters_ex(ZEND_PARSE_PARAMS_QUIET, ZEND_NUM_ARGS() | "sb", &callable, &do_throw, &prepend) == FAILURE) {
494:      return;
495:  }
496:  if (ZEND_NUM_ARGS() > 0) {
497:      if (!is_callable_ex(callable, NULL, IS_CALLABLE_STRICT, &func_name, &fci, &error)) {
498:          aif_func_ptr = fci.function_handler;
499:          aif_func_ptr = fci.function_handler;
500:          obj_ptr = fci.object;
501:          if (Z_TYPE_P(callable) == IS_ARRAY) {
502:              if (obj_ptr && aif_func_ptr && !aif_func_ptr->common.fn_flags & ZEND_ACC_STATIC) {
503:                  if (do_throw) {
504:                      zend_throw_exception_ex(spl_ce_LogicException, 0, "Passed array specifies a non static method but no object (\"%s\")", error);
505:                  }
506:              }
507:              if (error) {
508:                  efree(error);
509:              }
510:              zend_string_release(func_name);
511:              RETURN_FALSE;
512:          } else if (do_throw) {
513:              zend_throw_exception_ex(spl_ce_LogicException, 0, "Passed array does not specify to method (\"%s\")", aif_func_ptr ? "a callable" : "an exist
ng", obj_ptr ? "static" : "", error);
514:          }
515:          if (error) {
516:              efree(error);
517:          }
518:          zend_string_release(func_name);
519:          RETURN_FALSE;
520:      } else if (Z_TYPE_P(callable) == IS_STRING) {
521:          if (do_throw) {
522:              zend_throw_exception_ex(spl_ce_LogicException, 0, "Function '%s' not to (\"%s\")", ZSTR_VAL(func_name), aif_func_ptr ? "callable" : "found", er
or);
523:          }
524:          if (error) {
525:              efree(error);
526:          }
527:          zend_string_release(func_name);
528:          RETURN_FALSE;
529:      } else {
530:          if (do_throw) {
531:              zend_throw_exception_ex(spl_ce_LogicException, 0, "Illegal value passed (\"%s\")", error);
532:          }
533:          if (error) {
534:              efree(error);
535:          }
536:          zend_string_release(func_name);
537:          RETURN_FALSE;
538:      }
539:      if (fci.function_handler->type == ZEND_INTERNAL_FUNCTION &&
540:          fci.function_handler->internal_function_handler == &f_spl_autoload_call) {
541:          if (do_throw) {
542:              zend_throw_exception_ex(spl_ce_LogicException, 0, "Function spl_autoload_call() cannot be registered");
543:          }
544:          if (error) {
545:              efree(error);
546:          }
547:          zend_string_release(func_name);
548:          RETURN_FALSE;
549:      }
550:      aif_func_ptr = fci.function_handler;
551:      aif_func_ptr = fci.function_handler;
552:      obj_ptr = fci.object;
553:      if (error) {
554:          efree(error);
555:      }
556:  }
557:  if (Z_TYPE_P(callable) == IS_OBJECT) {
558:      ZVAL_COPY(&aif_closure, callable);
559:  }
560:  lc_name = zend_string_tolower(ZSTR_VAL(func_name));
561:  zend_ptr_dtor(&aif_closure);
562:  memcpy(ZSTR_VAL(lc_name) + ZSTR_LEN(func_name), &obj_ptr->handle, sizeof(uint32_t));

```



```

751:     return;
752: }
753: RETURN_FALSE;
754: }
755:
756: fptr = zend_hash_str_find_ptr(EG(function_table), "spl_autoload_call", sizeof("spl_autoload_call") - 1);
757:
758: if (EG(autoload_func) == fptr) {
759:     zend_string *key;
760:     array_init(&return_value);
761:     ZEND_HASH_FOREACH_STR_KEY_PTR(SPL_G(autoload_functions), key, aif) {
762:         if (!ZENDREF(aif->closure)) {
763:             Z_ADDREF(aif->closure);
764:             add_next_index_val(&return_value, &aif->closure);
765:         } else if (aif->func_ptr->common.scope) {
766:             zval tmp;
767:
768:             array_init(&tmp);
769:             if (!ZENDREF(aif->obj)) {
770:                 Z_ADDREF(aif->obj);
771:                 add_next_index_val(&tmp, &aif->obj);
772:             } else {
773:                 add_next_index_str(&tmp, zend_string_copy(aif->ce->name));
774:             }
775:             add_next_index_str(&tmp, zend_string_copy(aif->func_ptr->common.function_name));
776:             add_next_index_val(&return_value, &tmp);
777:         } else {
778:             if (ZENDREF(aif->func_ptr->common.function_name), "_lambda_func", &lambda_func" - 1)) {
779:                 add_next_index_str(&return_value, zend_string_copy(aif->func_ptr->common.function_name));
780:             } else {
781:                 add_next_index_str(&return_value, zend_string_copy(key));
782:             }
783:         }
784:     } ZEND_HASH_FOREACH_END();
785:     return;
786: }
787:
788: array_init(&return_value);
789: add_next_index_str(&return_value, zend_string_copy(EG(autoload_func)->common.function_name));
790: /* }}} */
791:
792: /* {{{ proto string spl_object_hash(object obj)
793:  * Return hash id for given object */
794: PHP_FUNCTION(spl_object_hash)
795: {
796:     zval *obj;
797:
798:     if (zend_parse_parameters(ZEND_NUM_ARGS() & "o", &obj) == FAILURE) {
799:         return;
800:     }
801:
802:     RETURN_NEW_STR(PHP_SPL_OBJECT_HASH(obj));
803: }
804: /* }}} */
805:
806: /* {{{ proto int spl_object_id(object obj)
807:  * Returns the integer object handle for the given object */
808: PHP_FUNCTION(spl_object_id)
809: {
810:     zval *obj;
811:
812:     ZEND_PARSE_PARAMETERS_START(1, 1)
813:     Z_PARAM_OBJECT(obj)
814:     ZEND_PARSE_PARAMETERS_END();
815:
816:     RETURN_LONG((zend_long)Z_OBJ_HANDLE_P(obj));
817: }
818: /* }}} */
819:
820: PHPAPI zend_string *php_spl_object_hash(zval *obj) /* {{{ */
821: {
822:     intptr_t hash_handle, hash_handlers;
823:
824:     if (!SPL_G(hash_mask_init)) {
825:         SPL_G(hash_mask_handle) = (intptr_t)(php_mt_rand() >> 1);
826:         SPL_G(hash_mask_handlers) = (intptr_t)(php_mt_rand() >> 1);
827:         SPL_G(hash_mask_init) = 1;
828:     }
829:
830:     hash_handle = SPL_G(hash_mask_handle) * (intptr_t)Z_OBJ_HANDLE_P(obj);
831:     hash_handlers = SPL_G(hash_mask_handlers);
832:
833:     return strprintf(32, "%06x%06x", hash_handle, hash_handlers);
834: }
835: /* }}} */
836:
837: int spl_build_class_list_string(zval *entry, char **list) /* {{{ */
838: {
839:     char *res;
840:
841:     sprintf(res, 0, "%s", *list, Z_STWAL_P(entry));
842:     zfree(*list);
843:     *list = res;
844:     return ZEND_HASH_APPLY_KEEP;
845: } /* }}} */
846:
847: /* {{{ PHP_MININFO(spl)
848: */
849: PHP_MININFO_FUNCTION(spl)
850: {
851:     zval list;
852:     char *str;
853:
854:     php_info_print_table_start();
855:     php_info_print_table_header(2, "SPL support", "enabled");
856:
857:     array_init(&list);
858:     SPL_LIST_CLASSES(&list, 0, 1, ZEND_ACC_INTERFACE);
859:     str = estrdup("");
860:     zend_hash_apply_with_argument(Z_ARRVAL_P(&list), (apply_func_arg_t)spl_build_class_list_string, &str);
861:     zval_dtor(&list);
862:     php_info_print_table_row(2, "Interfaces", str + 2);
863:     zfree(str);
864:
865:     array_init(&list);
866:     SPL_LIST_CLASSES(&list, 0, -1, ZEND_ACC_INTERFACE);
867:     str = estrdup("");
868:     zend_hash_apply_with_argument(Z_ARRVAL_P(&list), (apply_func_arg_t)spl_build_class_list_string, &str);
869:     zval_dtor(&list);
870:     php_info_print_table_row(2, "Classes", str + 2);
871:     zfree(str);
872:
873:     php_info_print_table_end();
874: }
875: /* }}} */
876:
877: /* {{{ arginfo */
878: ZEND_BEGIN_ARG_INFO_KK(arginfo_iterator_to_array, 0, 0, 1)
879:     ZEND_ARG_CBO_INFO(0, iterator, Traversable, 0)
880:     ZEND_ARG_INFO(0, use_keys)
881: ZEND_END_ARG_INFO();
882:
883: ZEND_BEGIN_ARG_INFO(arginfo_iterator, 0)
884:     ZEND_ARG_CBO_INFO(0, iterator, Traversable, 0)
885: ZEND_END_ARG_INFO();
886:
887: ZEND_BEGIN_ARG_INFO_KK(arginfo_iterator_apply, 0, 0, 2)
888:     ZEND_ARG_CBO_INFO(0, iterator, Traversable, 0)
889:     ZEND_ARG_INFO(0, function)
890:     ZEND_ARG_ARRAY_INFO(0, args, 1)
891: ZEND_END_ARG_INFO();
892:
893: ZEND_BEGIN_ARG_INFO_KK(arginfo_class_parents, 0, 0, 1)
894:     ZEND_ARG_INFO(0, instance)
895:     ZEND_ARG_INFO(0, autoload)
896: ZEND_END_ARG_INFO();
897:
898: ZEND_BEGIN_ARG_INFO_KK(arginfo_class_implements, 0, 0, 1)
899:     ZEND_ARG_INFO(0, what)
900:     ZEND_ARG_INFO(0, autoload)
901: ZEND_END_ARG_INFO();
902:
903: ZEND_BEGIN_ARG_INFO_KK(arginfo_class_uses, 0, 0, 1)
904:     ZEND_ARG_INFO(0, what)
905:     ZEND_ARG_INFO(0, autoload)
906: ZEND_END_ARG_INFO();
907:
908:
909: ZEND_BEGIN_ARG_INFO(arginfo_spl_classes, 0)
910:     ZEND_END_ARG_INFO();
911:
912: ZEND_BEGIN_ARG_INFO(arginfo_spl_autoload_functions, 0)
913:     ZEND_END_ARG_INFO();
914:
915: ZEND_BEGIN_ARG_INFO_KK(arginfo_spl_autoload, 0, 0, 1)
916:     ZEND_ARG_INFO(0, class_name)
917:     ZEND_ARG_INFO(0, file_extensions)
918: ZEND_END_ARG_INFO();
919:
920: ZEND_BEGIN_ARG_INFO_KK(arginfo_spl_autoload_extensions, 0, 0, 0)
921:     ZEND_ARG_INFO(0, file_extensions)
922: ZEND_END_ARG_INFO();
923:
924: ZEND_BEGIN_ARG_INFO_KK(arginfo_spl_autoload_call, 0, 0, 1)
925:     ZEND_ARG_INFO(0, class_name)
926: ZEND_END_ARG_INFO();
927:
928: ZEND_BEGIN_ARG_INFO_KK(arginfo_spl_autoload_register, 0, 0, 0)
929:     ZEND_ARG_INFO(0, autoload_function)
930:     ZEND_ARG_INFO(0, throw)
931:     ZEND_ARG_INFO(0, prepend)
932: ZEND_END_ARG_INFO();
933:
934: ZEND_BEGIN_ARG_INFO_KK(arginfo_spl_autoload_unregister, 0, 0, 1)
935:     ZEND_ARG_INFO(0, autoload_function)
936: ZEND_END_ARG_INFO();
937:
938: ZEND_BEGIN_ARG_INFO_KK(arginfo_spl_object_hash, 0, 0, 1)

```

```

939:     ZEND_ARG_INFO(0, obj)
940: ZEND_END_ARG_INFO();
941:
942: ZEND_BEGIN_ARG_INFO_KK(arginfo_spl_object_id, 0, 0, 1)
943:     ZEND_ARG_INFO(0, obj)
944: ZEND_END_ARG_INFO();
945: /* }}} */
946:
947: /* {{{ spl_functions
948: */
949: static const zend_function_entry spl_functions[] = {
950:     PHP_FE(spl_classes, arginfo_spl_classes)
951:     PHP_FE(spl_autoload, arginfo_spl_autoload)
952:     PHP_FE(spl_autoload_extensions, arginfo_spl_autoload_extensions)
953:     PHP_FE(spl_autoload_register, arginfo_spl_autoload_register)
954:     PHP_FE(spl_autoload_unregister, arginfo_spl_autoload_unregister)
955:     PHP_FE(spl_autoload_functions, arginfo_spl_autoload_functions)
956:     PHP_FE(spl_autoload_call, arginfo_spl_autoload_call)
957:     PHP_FE(class_parents, arginfo_class_parents)
958:     PHP_FE(class_implements, arginfo_class_implements)
959:     PHP_FE(class_uses, arginfo_class_uses)
960:     PHP_FE(spl_object_hash, arginfo_spl_object_hash)
961:     PHP_FE(spl_object_id, arginfo_spl_object_id)
962: #ifdef SPL_ITERATORS_H
963:     PHP_FE(iterator_to_array, arginfo_iterator_to_array)
964:     PHP_FE(iterator_count, arginfo_iterator)
965:     PHP_FE(iterator_apply, arginfo_iterator_apply)
966: #endif /* SPL_ITERATORS_H */
967:     PHP_FE_END
968: };
969: /* }}} */
970:
971: /* {{{ PHP_MINIT_FUNCTION(spl)
972: */
973: PHP_MINIT_FUNCTION(spl)
974: {
975:     PHP_MINIT(spl_exceptions) (INIT_FUNC_ARGS_PASSTHRU);
976:     PHP_MINIT(spl_iterators) (INIT_FUNC_ARGS_PASSTHRU);
977:     PHP_MINIT(spl_array) (INIT_FUNC_ARGS_PASSTHRU);
978:     PHP_MINIT(spl_directory) (INIT_FUNC_ARGS_PASSTHRU);
979:     PHP_MINIT(spl_dlist) (INIT_FUNC_ARGS_PASSTHRU);
980:     PHP_MINIT(spl_heap) (INIT_FUNC_ARGS_PASSTHRU);
981:     PHP_MINIT(spl_directory) (INIT_FUNC_ARGS_PASSTHRU);
982:     PHP_MINIT(spl_observer) (INIT_FUNC_ARGS_PASSTHRU);
983:
984:     return SUCCESS;
985: }
986: /* }}} */
987:
988: PHP_RINIT_FUNCTION(spl) /* {{{ */
989: {
990:     SPL_G(autoload_extensions) = NULL;
991:     SPL_G(autoload_functions) = NULL;
992:     SPL_G(hash_mask_init) = 0;
993:     return SUCCESS;
994: } /* }}} */
995:
996: PHP_SHUTDOWN_FUNCTION(spl) /* {{{ */
997: {
998:     if (SPL_G(autoload_extensions)) {
999:         zend_string_release(SPL_G(autoload_extensions));
1000:         SPL_G(autoload_extensions) = NULL;
1001:     }
1002:     if (SPL_G(autoload_functions)) {
1003:         zend_hash_destroy(SPL_G(autoload_functions));
1004:         FREE_HASHTABLE(SPL_G(autoload_functions));
1005:         SPL_G(autoload_functions) = NULL;
1006:     }
1007:     if (SPL_G(hash_mask_init)) {
1008:         SPL_G(hash_mask_init) = 0;
1009:     }
1010:     return SUCCESS;
1011: } /* }}} */
1012:
1013: /* {{{ spl_module_entry
1014: */
1015: zend_module_entry spl_module_entry = {
1016:     STANDARD_MODULE_HEADER,
1017:     "spl",
1018:     spl_functions,
1019:     PHP_MINIT(spl),
1020:     NULL,
1021:     PHP_RINIT(spl),
1022:     PHP_SHUTDOWN(spl),
1023:     PHP_MININFO(spl),
1024:     PHP_SPL_VERSION,
1025:     PHP_MODULE_GLOBALS(spl),
1026:     PHP_GINIT(spl),
1027:     NULL,
1028:     NULL,
1029:     STANDARD_MODULE_PROPERTIES_EX
1030: };
1031: /* }}} */
1032:
1033: /*
1034:  * Local variables:
1035:  * tab-width: 4
1036:  * c-basic-offset: 4
1037:  * End:
1038:  * vim600: fdm=marker
1039:  * vim: noet sw=4 ts=4
1040:  */

```

```
1: /*
2:  * -----
3:  * | PHP Version 7 |
4:  * -----
5:  * | Copyright (c) 1997-2018 The PHP Group |
6:  * -----
7:  * | This source file is subject to version 3.01 of the PHP license, |
8:  * | that is bundled with this package in the file LICENSE, and is |
9:  * | available through the world-wide-web at the following url: |
10:  * | http://www.php.net/license/3.01.txt |
11:  * | If you did not receive a copy of the PHP license and are unable to |
12:  * | obtain it through the world-wide-web, please send a note to |
13:  * | license@php.net so we can mail you a copy immediately. |
14:  * -----
15:  * | Authors: Marcus Boerger <helly@php.net> |
16:  * -----
17:  */
18:
19: /* $Id$ */
20:
21: #ifdef HAVE_CONFIG_H
22: #include "config.h"
23: #endif
24:
25: #include "php.h"
26: #include "php_ini.h"
27: #include "ext/standard/info.h"
28: #include "zend_interfaces.h"
29: #include "zend_exceptions.h"
30:
31: #include "spl_spl.h"
32: #include "spl_functions.h"
33: #include "spl_engine.h"
34: #include "spl_exceptions.h"
35:
36: PHPAPI zend_class_entry *spl_ce_LogicException;
37: PHPAPI zend_class_entry *spl_ce_BadFunctionCallException;
38: PHPAPI zend_class_entry *spl_ce_BadMethodCallException;
39: PHPAPI zend_class_entry *spl_ce_DomainException;
40: PHPAPI zend_class_entry *spl_ce_InvalidArgumentException;
41: PHPAPI zend_class_entry *spl_ce_LengthException;
42: PHPAPI zend_class_entry *spl_ce_OutOfRangeException;
43: PHPAPI zend_class_entry *spl_ce_RuntimeException;
44: PHPAPI zend_class_entry *spl_ce_OutOfBoundsException;
45: PHPAPI zend_class_entry *spl_ce_OverflowException;
46: PHPAPI zend_class_entry *spl_ce_RangeException;
47: PHPAPI zend_class_entry *spl_ce_UnderflowException;
48: PHPAPI zend_class_entry *spl_ce_UnexpectedValueException;
49:
50: #define spl_ce_Exception zend_ce_exception
51:
52: /* {{{ PHP_MINIT_FUNCTION(spl_exceptions) */
53: PHP_MINIT_FUNCTION(spl_exceptions)
54: {
55:     REGISTER_SPL_SUB_CLASS_EX(LogicException, Exception, NULL, NULL);
56:     REGISTER_SPL_SUB_CLASS_EX(BadFunctionCallException, LogicException, NULL, NULL);
57:     REGISTER_SPL_SUB_CLASS_EX(BadMethodCallException, BadFunctionCallException, NULL, NULL);
58:     REGISTER_SPL_SUB_CLASS_EX(DomainException, LogicException, NULL, NULL);
59:     REGISTER_SPL_SUB_CLASS_EX(InvalidArgumentException, LogicException, NULL, NULL);
60:     REGISTER_SPL_SUB_CLASS_EX(LengthException, LogicException, NULL, NULL);
61:     REGISTER_SPL_SUB_CLASS_EX(OutOfRangeException, LogicException, NULL, NULL);
62:
63:     REGISTER_SPL_SUB_CLASS_EX(RuntimeException, Exception, NULL, NULL);
64:     REGISTER_SPL_SUB_CLASS_EX(OutOfBoundsException, RuntimeException, NULL, NULL);
65:     REGISTER_SPL_SUB_CLASS_EX(OverflowException, RuntimeException, NULL, NULL);
66:     REGISTER_SPL_SUB_CLASS_EX(RangeException, RuntimeException, NULL, NULL);
67:     REGISTER_SPL_SUB_CLASS_EX(UnderflowException, RuntimeException, NULL, NULL);
68:     REGISTER_SPL_SUB_CLASS_EX(UnexpectedValueException, RuntimeException, NULL, NULL);
69:
70:     return SUCCESS;
71: }
72: /* }}} */
73:
74: /*
75:  * Local variables:
76:  * tab-width: 4
77:  * c-basic-offset: 4
78:  * End:
79:  * vim600: fdm=marker
80:  * vim: noet sw=4 ts=4
81:  */
```

```
1: /*
2:  * -----
3:  * | PHP Version 7 |
4:  * -----
5:  * | Copyright (c) 1997-2018 The PHP Group |
6:  * -----
7:  * | This source file is subject to version 3.01 of the PHP license, |
8:  * | that is bundled with this package in the file LICENSE, and is |
9:  * | available through the world-wide-web at the following url: |
10:  * | http://www.php.net/license/3.01.txt |
11:  * | If you did not receive a copy of the PHP license and are unable to |
12:  * | obtain it through the world-wide-web, please send a note to |
13:  * | license@php.net so we can mail you a copy immediately. |
14:  * -----
15:  * | Authors: Marcus Boerger <helly@php.net> |
16:  * -----
17:  */
18:
19: /* $Id$ */
20:
21: #ifndef SPL_OBSERVER_H
22: #define SPL_OBSERVER_H
23:
24: #include "php.h"
25: #include "php_spl.h"
26:
27: extern PHPAPI zend_class_entry *spl_ce_SplObserver;
28: extern PHPAPI zend_class_entry *spl_ce_SplSubject;
29: extern PHPAPI zend_class_entry *spl_ce_SplObjectStorage;
30: extern PHPAPI zend_class_entry *spl_ce_Multiplaiterator;
31:
32: PHP_MINIT_FUNCTION(spl_observer);
33:
34: #endif /* SPL_OBSERVER_H */
35:
36: /*
37:  * Local Variables:
38:  * c-basic-offset: 4
39:  * tab-width: 4
40:  * End:
41:  * vim600: fdm=marker
42:  * vim: noet sw=4 ts=4
43:  */
```

```
1: /*
2:  * -----
3:  * | PHP Version 7 |
4:  * -----
5:  * | Copyright (c) 1997-2018 The PHP Group |
6:  * -----
7:  * | This source file is subject to version 3.01 of the PHP license, |
8:  * | that is bundled with this package in the file LICENSE, and is |
9:  * | available through the world-wide-web at the following url: |
10:  * | http://www.php.net/license/3.01.txt |
11:  * | If you did not receive a copy of the PHP license and are unable to |
12:  * | obtain it through the world-wide-web, please send a note to |
13:  * | license@php.net so we can mail you a copy immediately. |
14:  * -----
15:  * | Authors: Marcus Boerger <helly@php.net> |
16:  * -----
17:  */
18:
19: /* $Id$ */
20:
21: #ifndef SPL_EXCEPTIONS_H
22: #define SPL_EXCEPTIONS_H
23:
24: #include "php.h"
25: #include "php_spl.h"
26:
27: extern PHPAPI zend_class_entry *spl_ce_LogicalException;
28: extern PHPAPI zend_class_entry *spl_ce_BadFunctionCallException;
29: extern PHPAPI zend_class_entry *spl_ce_BadMethodCallException;
30: extern PHPAPI zend_class_entry *spl_ce_DomainException;
31: extern PHPAPI zend_class_entry *spl_ce_InvalidArgumentException;
32: extern PHPAPI zend_class_entry *spl_ce_LengthException;
33: extern PHPAPI zend_class_entry *spl_ce_OutOfRangeException;
34:
35: extern PHPAPI zend_class_entry *spl_ce_RuntimeException;
36: extern PHPAPI zend_class_entry *spl_ce_OutOfBoundsException;
37: extern PHPAPI zend_class_entry *spl_ce_OverflowException;
38: extern PHPAPI zend_class_entry *spl_ce_RangeException;
39: extern PHPAPI zend_class_entry *spl_ce_UnderflowException;
40: extern PHPAPI zend_class_entry *spl_ce_UnexpectedValueException;
41:
42: PHP_MINIT_FUNCTION(spl_exceptions);
43:
44: #endif /* SPL_EXCEPTIONS_H */
45:
46: /*
47:  * Local Variables:
48:  * c-basic-offset: 4
49:  * tab-width: 4
50:  * End:
51:  * vim600: fdm=marker
52:  * vim: noet sw=4 ts=4
53:  */
```

```
1: /*
2:  * =====
3:  * | PHP Version 7 |
4:  * =====
5:  * | Copyright (c) 1997-2018 The PHP Group |
6:  * =====
7:  * | This source file is subject to version 3.01 of the PHP license, |
8:  * | that is bundled with this package in the file LICENSE, and is |
9:  * | available through the world-wide-web at the following url: |
10:  * | http://www.php.net/license/3.01.txt |
11:  * | If you did not receive a copy of the PHP license and are unable to |
12:  * | obtain it through the world-wide-web, please send a note to |
13:  * | license@php.net so we can mail you a copy immediately. |
14:  * =====
15:  * | Authors: Marcus Boerger <helly@php.net> |
16:  * =====
17:  */
18:
19: /* $Id$ */
20:
21: #ifndef HAVE_CONFIG_H
22: #include "config.h"
23: #endif
24:
25: #include "php.h"
26: #include "php_ini.h"
27: #include "ext/standard/info.h"
28: #include "php_spl.h"
29:
30: /* {{{ spl_register_interface */
31: void spl_register_interface zend_class_entry ** ppcw, char * class_name, const zend_function_entry * functions)
32: {
33:     zend_class_entry ce;
34:
35:     INIT_CLASS_ENTRY_EX(ce, class_name, strlen(class_name), functions);
36:     *ppcw = zend_register_internal_interface(&ce);
37: }
38: /* }}} */
39:
40: /* {{{ spl_register_std_class */
41: PHPAPI void spl_register_std_class(zend_class_entry ** ppcw, char * class_name, void * obj_ctor, const zend_function_entry * function_list)
42: {
43:     zend_class_entry ce;
44:
45:     INIT_CLASS_ENTRY_EX(ce, class_name, strlen(class_name), function_list);
46:     *ppcw = zend_register_internal_class(&ce);
47:
48:     /* entries changed by initialize */
49:     if (obj_ctor) {
50:         (*ppcw)->create_object = obj_ctor;
51:     }
52: }
53: /* }}} */
54:
55: /* {{{ spl_register_sub_class */
56: PHPAPI void spl_register_sub_class(zend_class_entry ** ppcw, zend_class_entry * parent_ce, char * class_name, void * obj_ctor, const zend_function_entry * function_list)
57: {
58:     zend_class_entry ce;
59:
60:     INIT_CLASS_ENTRY_EX(ce, class_name, strlen(class_name), function_list);
61:     *ppcw = zend_register_internal_class_ex(&ce, parent_ce);
62:
63:     /* entries changed by initialize */
64:     if (obj_ctor) {
65:         (*ppcw)->create_object = obj_ctor;
66:     } else {
67:         (*ppcw)->create_object = parent_ce->create_object;
68:     }
69: }
70: /* }}} */
71:
72: /* {{{ spl_register_property */
73: void spl_register_property(zend_class_entry * class_entry, char *prop_name, int prop_name_len, int prop_flags)
74: {
75:     zend_declare_property_null(class_entry, prop_name, prop_name_len, prop_flags);
76: }
77: /* }}} */
78:
79: /* {{{ spl_add_class_name */
80: void spl_add_class_name(zval *list, zend_class_entry *pcw, int allow, int ce_flags)
81: {
82:     if ((allow && (allow > 0 && pcw->ce_flags & ce_flags)) || (allow < 0 && !(pcw->ce_flags & ce_flags))) {
83:         zval *tmp;
84:
85:         if ((tmp = zend_hash_find(Z_ARRVAL_P(list), pcw->name)) == NULL) {
86:             zval tmp;
87:             ZVAL_STR_COPY(&tmp, pcw->name);
88:             zend_hash_add(Z_ARRVAL_P(list), pcw->name, &tmp);
89:         }
90:     }
91: }
92: /* }}} */
93:
94: /* {{{ spl_add_interfaces */
95: void spl_add_interfaces(zval *list, zend_class_entry * pcw, int allow, int ce_flags)
96: {
97:     uint32_t num_interfaces;
98:
99:     for (num_interfaces = 0; num_interfaces < pcw->num_interfaces; num_interfaces++) {
100:         spl_add_class_name(list, pcw->interfaces[num_interfaces], allow, ce_flags);
101:     }
102: }
103: /* }}} */
104:
105: /* {{{ spl_add_traits */
106: void spl_add_traits(zval *list, zend_class_entry * pcw, int allow, int ce_flags)
107: {
108:     uint32_t num_traits;
109:
110:     for (num_traits = 0; num_traits < pcw->num_traits; num_traits++) {
111:         spl_add_class_name(list, pcw->traits[num_traits], allow, ce_flags);
112:     }
113: }
114: /* }}} */
115:
116: /* {{{ spl_add_classes */
117: int spl_add_classes(zend_class_entry *pcw, zval *list, int sub, int allow, int ce_flags)
118: {
119:     if (!pcw) {
120:         return 0;
121:     }
122:     spl_add_class_name(list, pcw, allow, ce_flags);
123:     if (sub) {
124:         spl_add_interfaces(list, pcw, allow, ce_flags);
125:         while (pcw->parent) {
126:             pcw = pcw->parent;
127:             spl_add_classes(pcw, list, sub, allow, ce_flags);
128:         }
129:     }
130:     return 0;
131: }
132: /* }}} */
133:
134: zend_string * spl_get_private_prop_name(zend_class_entry *ce, char *prop_name, int prop_len) /* {{{ */
135: {
136:     return zend_mangle_property_name(ZSTR_VAL(ce->name), ZSTR_LEN(ce->name), prop_name, prop_len, 0);
137: }
138: /* }}} */
139:
140: /*
141:  * Local Variables:
142:  * tab-width: 4
143:  * c-basic-offset: 4
144:  * End:
145:  * vim600: fdm=marker
146:  * vim: noet sw=4 ts=4
147:  */
148: */
```

```
1: /*
2:  *
3:  * PHP Version 7
4:  *
5:  * Copyright (c) 1997-2018 The PHP Group
6:  *
7:  * This source file is subject to version 3.01 of the PHP license,
8:  * that is bundled with this package in the file LICENSE, and is
9:  * available through the world-wide-web at the following url:
10:  * https://www.php.net/license/3.01.txt
11:  * If you did not receive a copy of the PHP license and are unable to
12:  * obtain it through the world-wide-web, please send a note to
13:  * license@php.net so we can mail you a copy immediately.
14:  *
15:  * Authors: Marcus Boerger <helly@php.net>
16:  *
17:  */
18:
19: /* $Id$ */
20:
21: #ifndef SPL_DIRECTORY_H
22: #define SPL_DIRECTORY_H
23:
24: #include "php.h"
25: #include "php_spl.h"
26:
27: extern PHPAPI zend_class_entry *spl_ce_SplFileInfo;
28: extern PHPAPI zend_class_entry *spl_ce_DirectoryIterator;
29: extern PHPAPI zend_class_entry *spl_ce_FilesystemIterator;
30: extern PHPAPI zend_class_entry *spl_ce_RecursiveDirectoryIterator;
31: extern PHPAPI zend_class_entry *spl_ce_GlobIterator;
32: extern PHPAPI zend_class_entry *spl_ce_SplFileInfo;
33: extern PHPAPI zend_class_entry *spl_ce_SplTempFileObject;
34:
35: PHP_MINIT_FUNCTION(spl_directory);
36:
37: typedef enum {
38:     SPL_FS_INFO, /* must be 0 */
39:     SPL_FS_DIR,
40:     SPL_FS_FILE,
41: } SPL_FS_OBJ_TYPE;
42:
43: typedef struct _spl_filesystem_object spl_filesystem_object;
44:
45: typedef void (*spl_foreign_dtor_t)(spl_filesystem_object *object);
46: typedef void (*spl_foreign_clone_t)(spl_filesystem_object *src, spl_filesystem_object *dst);
47:
48: PHPAPI char* spl_filesystem_object_get_path(spl_filesystem_object *intern, size_t *len);
49:
50: typedef struct _spl_other_handler {
51:     spl_foreign_dtor_t dtor;
52:     spl_foreign_clone_t clone;
53: } spl_other_handler;
54:
55: /* define an overloaded iterator structure */
56: typedef struct {
57:     zend_object_iterator intern;
58:     zval *current;
59:     void *object;
60: } spl_filesystem_iterator;
61:
62: struct _spl_filesystem_object {
63:     void *other;
64:     const spl_other_handler *other_handler;
65:     char *_path;
66:     size_t _path_len;
67:     char *_full_path;
68:     char *_file_name;
69:     size_t file_name_len;
70:     SPL_FS_OBJ_TYPE type;
71:     zend_long flags;
72:     zend_class_entry *file_class;
73:     zend_class_entry *info_class;
74:     union {
75:         struct {
76:             php_stream *stream;
77:             php_stream_dirent *entry;
78:             char *sub_path;
79:             size_t sub_path_len;
80:             int index;
81:             int is_recursive;
82:             zend_function *func_read;
83:             zend_function *func_next;
84:             zend_function *func_valid;
85:         } dir;
86:         struct {
87:             php_stream *stream;
88:             php_stream_context *context;
89:             zval *scontext;
90:             char *open_mode;
91:             size_t open_mode_len;
92:             zval *current_zval;
93:             char *current_line;
94:             size_t current_line_len;
95:             size_t max_line_len;
96:             zend_long current_line_num;
97:             zval *resource;
98:             zend_function *func_getCurr;
99:             char *delimiter;
100:             char *enclosure;
101:             char *escape;
102:         } file;
103:     } u;
104:     zend_object std;
105: };
106:
107: static inline spl_filesystem_object *spl_filesystem_from_obj(zend_object *obj) /* {{{ */ {
108:     return (spl_filesystem_object *) ((char *) (obj) - XOffsetOf(spl_filesystem_object, std));
109: }
110: /* }}} */
111:
112: #define Z_SPL_FILESYSTEM_P(zv) spl_filesystem_from_obj(Z_OBJ_P(zv))
113:
114: static inline spl_filesystem_iterator *spl_filesystem_object_to_iterator(spl_filesystem_object *obj)
115: {
116:     spl_filesystem_iterator *it;
117:
118:     it = ecalloc(1, sizeof(spl_filesystem_iterator));
119:     it->object = (void *) obj;
120:     zend_iterator_init(it->intern);
121:     return it;
122: }
123:
124: static inline spl_filesystem_object *spl_filesystem_iterator_to_object(spl_filesystem_iterator *it)
125: {
126:     return (spl_filesystem_object *) it->object;
127: }
128:
129: #define SPL_FILE_OBJECT_DROP_NEW_LINE 0x00000001 /* drop new lines */
130: #define SPL_FILE_OBJECT_READ_HEADER 0x00000002 /* read or read/next */
131: #define SPL_FILE_OBJECT_SKIP_EMPTY 0x00000004 /* skip empty lines */
132: #define SPL_FILE_OBJECT_READ_CSV 0x00000008 /* read via fgets */
133: #define SPL_FILE_OBJECT_MASK 0x0000000F /* read via fgets */
134:
135: #define SPL_FILE_DIR_CURRENT_AS_FILEINFO 0x00000000 /* make RecursiveDirectoryTree::current() return SplFileInfo */
136: #define SPL_FILE_DIR_CURRENT_AS_ZIP 0x00000001 /* make RecursiveDirectoryTree::current() return getZip() */
137: #define SPL_FILE_DIR_CURRENT_AS_PATHNAME 0x00000020 /* make RecursiveDirectoryTree::current() return getPathname() */
138: #define SPL_FILE_DIR_CURRENT_MODE_MASK 0x000000F0 /* mask RecursiveDirectoryTree::current() */
139: #define SPL_FILE_DIR_CURRENT_MODE_MASK ((intern->flags & SPL_FILE_DIR_CURRENT_MODE_MASK) ~ mode)
140:
141: #define SPL_FILE_DIR_KEY_AS_PATHNAME 0x00000000 /* make RecursiveDirectoryTree::key() return getPathname() */
142: #define SPL_FILE_DIR_KEY_AS_PATHNAME 0x00000000 /* make RecursiveDirectoryTree::key() return getPathname() */
143: #define SPL_FILE_DIR_FOLLOW_SYMLINKS 0x00000020 /* make RecursiveDirectoryTree::hasChildren() follow symlinks */
144: #define SPL_FILE_DIR_KEY_MODE_MASK 0x000000F0 /* mask RecursiveDirectoryTree::key() */
145: #define SPL_FILE_DIR_KEY_MODE_MASK ((intern->flags & SPL_FILE_DIR_KEY_MODE_MASK) ~ mode)
146:
147: #define SPL_FILE_DIR_SKIPDOTS 0x00000100 /* Tells whether it should skip dots or not */
148: #define SPL_FILE_DIR_UNIFYPATHS 0x00000200 /* whether to unify path separators */
149: #define SPL_FILE_DIR_OTHERS_MASK 0x00000300 /* mask used for get/setFlags */
150:
151: #endif /* SPL_DIRECTORY_H */
152:
153: /*
154:  * Local Variables:
155:  * c-basic-offset: 4
156:  * tab-width: 4
157:  * End:
158:  * vim600: fdm=marker
159:  * vim: noet sw=4 ts=4
160:  */
```

```

1: /*
2:  * PHP Version 7
3:  *
4:  * Copyright (c) 1997-2018 The PHP Group
5:  *
6:  * This source file is subject to version 3.01 of the PHP license,
7:  * that is bundled with this package in the file LICENSE, and is
8:  * available through the world-wide-web at the following url:
9:  * http://www.php.net/license/3_01.txt
10:  * If you did not receive a copy of the PHP license and are unable to
11:  * obtain it through the world-wide-web, please send a note to
12:  * license@php.net so we can mail you a copy immediately.
13:  *
14:  * Author: Antony Duvall <tony@daylesday.org>
15:  *
16:  * Elienne Kneuss <elienne@php.net>
17:  */
18:
19:
20: /* $Id$ */
21:
22: #ifndef HAVE_CONFIG_H
23: #include "config.h"
24: #endif
25:
26: #include "php.h"
27: #include "php_ini.h"
28: #include "ext/standard/info.h"
29: #include "zend_exceptions.h"
30:
31: #include "spl_spl.h"
32: #include "spl_functions.h"
33: #include "spl_engine.h"
34: #include "spl_fixedarray.h"
35: #include "spl_exceptions.h"
36: #include "spl_iterators.h"
37:
38: zend_object_handlers spl_handler_SplFixedArray;
39: PHP_API zend_class_entry *spl_ce_SplFixedArray;
40:
41: #ifndef COMPILE_DL_SPL_FIXEDARRAY
42: #error GET_MODULE(spl_fixedarray)
43: #endif
44:
45: typedef struct _spl_fixedarray { /* {{{ */
46:     zend_long size;
47:     zval *elements;
48: } spl_fixedarray;
49: /* }}} */
50:
51: typedef struct _spl_fixedarray_object { /* {{{ */
52:     spl_fixedarray array;
53:     zval *offset_get;
54:     zend_function *fptr_offset_set;
55:     zend_function *fptr_offset_has;
56:     zend_function *fptr_offset_del;
57:     zend_function *fptr_count;
58:     int current;
59:     int flags;
60:     zend_class_entry *ce_iterator;
61:     zend_object std;
62: } spl_fixedarray_object;
63: /* }}} */
64:
65: typedef struct _spl_fixedarray_it { /* {{{ */
66:     zend_object_iterator intern;
67: } spl_fixedarray_it;
68: /* }}} */
69:
70: #define SPL_FIXEDARRAY_OVERLOADED_REWIND 0x0001
71: #define SPL_FIXEDARRAY_OVERLOADED_VALID 0x0002
72: #define SPL_FIXEDARRAY_OVERLOADED_KEY 0x0004
73: #define SPL_FIXEDARRAY_OVERLOADED_CURRENT 0x0008
74: #define SPL_FIXEDARRAY_OVERLOADED_NEXT 0x0010
75:
76: static inline spl_fixedarray_object *spl_fixed_array_from_obj(zend_object *obj) /* {{{ */ {
77:     return (spl_fixedarray_object *) ((char *) (obj) - XFF_OFFSET(spl_fixedarray_object, std));
78: }
79: /* }}} */
80:
81: #define SPL_FIXEDARRAY_P(rv) spl_fixed_array_from_obj(Z_OBJ_P(rv))
82:
83: static void spl_fixedarray_init(spl_fixedarray *array, zend_long size) /* {{{ */ {
84:     if (size > 0) {
85:         array->size = 0; /* reset size in case realloc() fails */
86:         array->elements = ecalloc(size, sizeof(zval));
87:         array->size = size;
88:     } else {
89:         array->elements = NULL;
90:         array->size = 0;
91:     }
92: }
93: /* }}} */
94:
95: static void spl_fixedarray_resize(spl_fixedarray *array, zend_long size) /* {{{ */ {
96:     if (size == array->size) {
97:         /* nothing to do */
98:         return;
99:     }
100:
101:     /* first initialization */
102:     if (array->size == 0) {
103:         spl_fixedarray_init(array, size);
104:         return;
105:     }
106:
107:     /* clearing the array */
108:     if (size == 0) {
109:         zend_long i;
110:
111:         for (i = 0; i < array->size; i++) {
112:             zval_ptr_dtor(&(array->elements[i]));
113:         }
114:
115:         if (array->elements) {
116:             zfree(array->elements);
117:             array->elements = NULL;
118:         }
119:     } else if (size > array->size) {
120:         array->elements = safe_realloc(array->elements, size, sizeof(zval), 0);
121:         memset(array->elements + array->size, '\0', sizeof(zval) * (size - array->size));
122:     } else { /* size < array->size */
123:         zend_long i;
124:
125:         for (i = size; i < array->size; i++) {
126:             zval_ptr_dtor(&(array->elements[i]));
127:         }
128:
129:         array->elements = erealloc(array->elements, sizeof(zval) * size);
130:     }
131:
132:     array->size = size;
133: }
134: /* }}} */
135:
136: static void spl_fixedarray_copy(spl_fixedarray *to, spl_fixedarray *from) /* {{{ */ {
137:     zend_long i;
138:
139:     for (i = 0; i < from->size; i++) {
140:         ZVAL_COPY(&(to->elements[i]), &(from->elements[i]));
141:     }
142: }
143: /* }}} */
144:
145: static HashTable* spl_fixedarray_object_get_ge(zval *obj, zval **table, int *n) /* {{{ */ {
146:     spl_fixedarray_object *intern = Z_SPLFIXEDARRAY_P(obj);
147:     HashTable *ht = zend_std_get_properties(obj);
148:
149:     *table = intern->array.elements;
150:     *n = (int) intern->array.size;
151:     return ht;
152: }
153: /* }}} */
154:
155: static HashTable* spl_fixedarray_object_get_properties(zval *obj) /* {{{ */ {
156:     spl_fixedarray_object *intern = Z_SPLFIXEDARRAY_P(obj);
157:     HashTable *ht = zend_std_get_properties(obj);
158:     zend_long i = 0;
159:
160:     if (intern->array.size > 0) {
161:         zend_long j = zend_hash_num_elements(ht);
162:
163:         for (i = 0; i < intern->array.size; i++) {
164:             if (IS_UNDEF(intern->array.elements[i])) {
165:                 zend_hash_index_update(ht, i, &(intern->array.elements[i]));
166:             }
167:             Z_STR_ASSIGN(&(intern->array.elements[i]));
168:         }
169:     } else {
170:         zend_hash_index_update(ht, 1, &(uninitialized_zval));
171:     }
172:
173:     return ht;
174: }
175:
176: if (j > intern->array.size) {
177:     for (i = intern->array.size; i < j; ++i) {
178:         zend_hash_index_del(ht, i);
179:     }
180: }
181:
182: return ht;
183: }
184: /* }}} */
185:
186: static void spl_fixedarray_object_free_storage(zend_object *obj) /* {{{ */ {
187:     spl_fixedarray_object *intern = spl_fixed_array_from_obj(obj);
188:
189:     zend_long i;
190:
191:     for (i = 0; i < intern->array.size; i++) {
192:         zval_ptr_dtor(&(intern->array.elements[i]));
193:     }
194:
195:     if (intern->array.elements) {
196:         zfree(intern->array.elements);
197:     }
198:
199:     zend_object_std_dtor((zend_object_std *) intern);
200:
201:     zend_object_std_init((zend_object_std *) intern, zend_class_entry *ce_iterator);
202:     object_properties_init(intern->std, class_type);
203:     intern->std->handlers = spl_handler_SplFixedArray;
204:     class_type->get_iterator = spl_fixedarray_get_iterator;
205:     break;
206: }
207:
208: while (parent) {
209:     if (parent == spl_ce_SplFixedArray) {
210:         intern->std->handlers = spl_handler_SplFixedArray;
211:         class_type->get_iterator = spl_fixedarray_get_iterator;
212:         break;
213:     }
214:     parent = parent->parent;
215:     inherited = 1;
216: }
217:
218: if (!parent) { /* this must never happen */
219:     php_error_fatal("Internal compiler error, Class is not child of SplFixedArray");
220: }
221:
222: if (class_type->iterator_funcs.if_rewind == zend_hash_str_find_ptr(class_type->function_table, "rewind", sizeof("rewind") - 1);
223:     class_type->iterator_funcs.if_valid == zend_hash_str_find_ptr(class_type->function_table, "valid", sizeof("valid") - 1);
224:     class_type->iterator_funcs.if_key == zend_hash_str_find_ptr(class_type->function_table, "key", sizeof("key") - 1);
225:     class_type->iterator_funcs.if_current == zend_hash_str_find_ptr(class_type->function_table, "current", sizeof("current") - 1);
226:     class_type->iterator_funcs.if_next == zend_hash_str_find_ptr(class_type->function_table, "next", sizeof("next") - 1);
227: }
228:
229: if (inherited) {
230:     if (class_type->iterator_funcs.if_rewind->common.scope != parent) {
231:         intern->flags |= SPL_FIXEDARRAY_OVERLOADED_REWIND;
232:     }
233:     if (class_type->iterator_funcs.if_valid->common.scope != parent) {
234:         intern->flags |= SPL_FIXEDARRAY_OVERLOADED_VALID;
235:     }
236:     if (class_type->iterator_funcs.if_key->common.scope != parent) {
237:         intern->flags |= SPL_FIXEDARRAY_OVERLOADED_KEY;
238:     }
239:     if (class_type->iterator_funcs.if_current->common.scope != parent) {
240:         intern->flags |= SPL_FIXEDARRAY_OVERLOADED_CURRENT;
241:     }
242:     if (class_type->iterator_funcs.if_next->common.scope != parent) {
243:         intern->flags |= SPL_FIXEDARRAY_OVERLOADED_NEXT;
244:     }
245:
246:     intern->fptr_offset_get = zend_hash_str_find_ptr(class_type->function_table, "offsetget", sizeof("offsetget") - 1);
247:     if (intern->fptr_offset_get->common.scope == parent) {
248:         intern->fptr_offset_get = NULL;
249:     }
250:     intern->fptr_offset_set = zend_hash_str_find_ptr(class_type->function_table, "offsetset", sizeof("offsetset") - 1);
251:     if (intern->fptr_offset_set->common.scope == parent) {
252:         intern->fptr_offset_set = NULL;
253:     }
254:     intern->fptr_offset_has = zend_hash_str_find_ptr(class_type->function_table, "offsetexists", sizeof("offsetexists") - 1);
255:     if (intern->fptr_offset_has->common.scope == parent) {
256:         intern->fptr_offset_has = NULL;
257:     }
258:     intern->fptr_offset_del = zend_hash_str_find_ptr(class_type->function_table, "offsetunset", sizeof("offsetunset") - 1);
259:     if (intern->fptr_offset_del->common.scope == parent) {
260:         intern->fptr_offset_del = NULL;
261:     }
262:     intern->fptr_count = zend_hash_str_find_ptr(class_type->function_table, "count", sizeof("count") - 1);
263:     if (intern->fptr_count->common.scope == parent) {
264:         intern->fptr_count = NULL;
265:     }
266: }
267:
268: return intern->std;
269: }
270: /* }}} */
271:
272: static zend_object *spl_fixedarray_new(zend_class_entry *class_type) /* {{{ */ {
273:     zend_object *obj = zend_object_alloc(sizeof(spl_fixedarray_object), class_type);
274:     obj->std->handlers = spl_handler_SplFixedArray;
275:     obj->std->ce = class_type;
276:     obj->std->flags = 0;
277:     obj->std->handlers = spl_handler_SplFixedArray;
278:     obj->std->ce = class_type;
279:     obj->std->flags = 0;
280:     obj->std->handlers = spl_handler_SplFixedArray;
281:     obj->std->ce = class_type;
282:     obj->std->flags = 0;
283:     obj->std->handlers = spl_handler_SplFixedArray;
284:     obj->std->ce = class_type;
285:     obj->std->flags = 0;
286:     obj->std->handlers = spl_handler_SplFixedArray;
287:     obj->std->ce = class_type;
288:     obj->std->flags = 0;
289:     obj->std->handlers = spl_handler_SplFixedArray;
290:     obj->std->ce = class_type;
291:     obj->std->flags = 0;
292:     obj->std->handlers = spl_handler_SplFixedArray;
293:     obj->std->ce = class_type;
294:     obj->std->flags = 0;
295:     obj->std->handlers = spl_handler_SplFixedArray;
296:     obj->std->ce = class_type;
297:     obj->std->flags = 0;
298:     obj->std->handlers = spl_handler_SplFixedArray;
299:     obj->std->ce = class_type;
300:     obj->std->flags = 0;
301:     obj->std->handlers = spl_handler_SplFixedArray;
302:     obj->std->ce = class_type;
303:     obj->std->flags = 0;
304:     obj->std->handlers = spl_handler_SplFixedArray;
305:     obj->std->ce = class_type;
306:     obj->std->flags = 0;
307:     obj->std->handlers = spl_handler_SplFixedArray;
308:     obj->std->ce = class_type;
309:     obj->std->flags = 0;
310:     obj->std->handlers = spl_handler_SplFixedArray;
311:     obj->std->ce = class_type;
312:     obj->std->flags = 0;
313:     obj->std->handlers = spl_handler_SplFixedArray;
314:     obj->std->ce = class_type;
315:     obj->std->flags = 0;
316:     obj->std->handlers = spl_handler_SplFixedArray;
317:     obj->std->ce = class_type;
318:     obj->std->flags = 0;
319:     obj->std->handlers = spl_handler_SplFixedArray;
320:     obj->std->ce = class_type;
321:     obj->std->flags = 0;
322:     obj->std->handlers = spl_handler_SplFixedArray;
323:     obj->std->ce = class_type;
324:     obj->std->flags = 0;
325:     obj->std->handlers = spl_handler_SplFixedArray;
326:     obj->std->ce = class_type;
327:     obj->std->flags = 0;
328:     obj->std->handlers = spl_handler_SplFixedArray;
329:     obj->std->ce = class_type;
330:     obj->std->flags = 0;
331:     obj->std->handlers = spl_handler_SplFixedArray;
332:     obj->std->ce = class_type;
333:     obj->std->flags = 0;
334:     obj->std->handlers = spl_handler_SplFixedArray;
335:     obj->std->ce = class_type;
336:     obj->std->flags = 0;
337:     obj->std->handlers = spl_handler_SplFixedArray;
338:     obj->std->ce = class_type;
339:     obj->std->flags = 0;
340:     obj->std->handlers = spl_handler_SplFixedArray;
341:     obj->std->ce = class_type;
342:     obj->std->flags = 0;
343:     obj->std->handlers = spl_handler_SplFixedArray;
344:     obj->std->ce = class_type;
345:     obj->std->flags = 0;
346:     obj->std->handlers = spl_handler_SplFixedArray;
347:     obj->std->ce = class_type;
348:     obj->std->flags = 0;
349:     obj->std->handlers = spl_handler_SplFixedArray;
350:     obj->std->ce = class_type;
351:     obj->std->flags = 0;
352:     obj->std->handlers = spl_handler_SplFixedArray;
353:     obj->std->ce = class_type;
354:     obj->std->flags = 0;
355:     obj->std->handlers = spl_handler_SplFixedArray;
356:     obj->std->ce = class_type;
357:     obj->std->flags = 0;
358:     obj->std->handlers = spl_handler_SplFixedArray;
359:     obj->std->ce = class_type;
360:     obj->std->flags = 0;
361:     obj->std->handlers = spl_handler_SplFixedArray;
362:     obj->std->ce = class_type;
363:     obj->std->flags = 0;
364:     obj->std->handlers = spl_handler_SplFixedArray;
365:     obj->std->ce = class_type;
366:     obj->std->flags = 0;
367:     obj->std->handlers = spl_handler_SplFixedArray;
368:     obj->std->ce = class_type;
369:     obj->std->flags = 0;
370:     obj->std->handlers = spl_handler_SplFixedArray;
371:     obj->std->ce = class_type;
372:     obj->std->flags = 0;
373:     obj->std->handlers = spl_handler_SplFixedArray;
374:     obj->std->ce = class_type;
375:     obj->std->flags = 0;
376:     obj->std->handlers = spl_handler_SplFixedArray;
377:     obj->std->ce = class_type;
378:     obj->std->flags = 0;
379:     obj->std->handlers = spl_handler_SplFixedArray;
380:     obj->std->ce = class_type;
381:     obj->std->flags = 0;
382:     obj->std->handlers = spl_handler_SplFixedArray;
383:     obj->std->ce = class_type;
384:     obj->std->flags = 0;
385:     obj->std->handlers = spl_handler_SplFixedArray;
386:     obj->std->ce = class_type;
387:     obj->std->flags = 0;
388:     obj->std->handlers = spl_handler_SplFixedArray;
389:     obj->std->ce = class_type;
390:     obj->std->flags = 0;
391:     obj->std->handlers = spl_handler_SplFixedArray;
392:     obj->std->ce = class_type;
393:     obj->std->flags = 0;
394:     obj->std->handlers = spl_handler_SplFixedArray;
395:     obj->std->ce = class_type;
396:     obj->std->flags = 0;
397:     obj->std->handlers = spl_handler_SplFixedArray;
398:     obj->std->ce = class_type;
399:     obj->std->flags = 0;
400:     obj->std->handlers = spl_handler_SplFixedArray;
401:     obj->std->ce = class_type;
402:     obj->std->flags = 0;
403:     obj->std->handlers = spl_handler_SplFixedArray;
404:     obj->std->ce = class_type;
405:     obj->std->flags = 0;
406:     obj->std->handlers = spl_handler_SplFixedArray;
407:     obj->std->ce = class_type;
408:     obj->std->flags = 0;
409:     obj->std->handlers = spl_handler_SplFixedArray;
410:     obj->std->ce = class_type;
411:     obj->std->flags = 0;
412:     obj->std->handlers = spl_handler_SplFixedArray;
413:     obj->std->ce = class_type;
414:     obj->std->flags = 0;
415:     obj->std->handlers = spl_handler_SplFixedArray;
416:     obj->std->ce = class_type;
417:     obj->std->flags = 0;
418:     obj->std->handlers = spl_handler_SplFixedArray;
419:     obj->std->ce = class_type;
420:     obj->std->flags = 0;
421:     obj->std->handlers = spl_handler_SplFixedArray;
422:     obj->std->ce = class_type;
423:     obj->std->flags = 0;
424:     obj->std->handlers = spl_handler_SplFixedArray;
425:     obj->std->ce = class_type;
426:     obj->std->flags = 0;
427:     obj->std->handlers = spl_handler_SplFixedArray;
428:     obj->std->ce = class_type;
429:     obj->std->flags = 0;
430:     obj->std->handlers = spl_handler_SplFixedArray;
431:     obj->std->ce = class_type;
432:     obj->std->flags = 0;
433:     obj->std->handlers = spl_handler_SplFixedArray;
434:     obj->std->ce = class_type;
435:     obj->std->flags = 0;
436:     obj->std->handlers = spl_handler_SplFixedArray;
437:     obj->std->ce = class_type;
438:     obj->std->flags = 0;
439:     obj->std->handlers = spl_handler_SplFixedArray;
440:     obj->std->ce = class_type;
441:     obj->std->flags = 0;
442:     obj->std->handlers = spl_handler_SplFixedArray;
443:     obj->std->ce = class_type;
444:     obj->std->flags = 0;
445:     obj->std->handlers = spl_handler_SplFixedArray;
446:     obj->std->ce = class_type;
447:     obj->std->flags = 0;
448:     obj->std->handlers = spl_handler_SplFixedArray;
449:     obj->std->ce = class_type;
450:     obj->std->flags = 0;
451:     obj->std->handlers = spl_handler_SplFixedArray;
452:     obj->std->ce = class_type;
453:     obj->std->flags = 0;
454:     obj->std->handlers = spl_handler_SplFixedArray;
455:     obj->std->ce = class_type;
456:     obj->std->flags = 0;
457:     obj->std->handlers = spl_handler_SplFixedArray;
458:     obj->std->ce = class_type;
459:     obj->std->flags = 0;
460:     obj->std->handlers = spl_handler_SplFixedArray;
461:     obj->std->ce = class_type;
462:     obj->std->flags = 0;
463:     obj->std->handlers = spl_handler_SplFixedArray;
464:     obj->std->ce = class_type;
465:     obj->std->flags = 0;
466:     obj->std->handlers = spl_handler_SplFixedArray;
467:     obj->std->ce = class_type;
468:     obj->std->flags = 0;
469:     obj->std->handlers = spl_handler_SplFixedArray;
470:     obj->std->ce = class_type;
471:     obj->std->flags = 0;
472:     obj->std->handlers = spl_handler_SplFixedArray;
473:     obj->std->ce = class_type;
474:     obj->std->flags = 0;
475:     obj->std->handlers = spl_handler_SplFixedArray;
476:     obj->std->ce = class_type;
477:     obj->std->flags = 0;
478:     obj->std->handlers = spl_handler_SplFixedArray;
479:     obj->std->ce = class_type;
480:     obj->std->flags = 0;
481:     obj->std->handlers = spl_handler_SplFixedArray;
482:     obj->std->ce = class_type;
483:     obj->std->flags = 0;
484:     obj->std->handlers = spl_handler_SplFixedArray;
485:     obj->std->ce = class_type;
486:     obj->std->flags = 0;
487:     obj->std->handlers = spl_handler_SplFixedArray;
488:     obj->std->ce = class_type;
489:     obj->std->flags = 0;
490:     obj->std->handlers = spl_handler_SplFixedArray;
491:     obj->std->ce = class_type;
492:     obj->std->flags = 0;
493:     obj->std->handlers = spl_handler_SplFixedArray;
494:     obj->std->ce = class_type;
495:     obj->std->flags = 0;
496:     obj->std->handlers = spl_handler_SplFixedArray;
497:     obj->std->ce = class_type;
498:     obj->std->flags = 0;
499:     obj->std->handlers = spl_handler_SplFixedArray;
500:     obj->std->ce = class_type;
501:     obj->std->flags = 0;
502:     obj->std->handlers = spl_handler_SplFixedArray;
503:     obj->std->ce = class_type;
504:     obj->std->flags = 0;
505:     obj->std->handlers = spl_handler_SplFixedArray;
506:     obj->std->ce = class_type;
507:     obj->std->flags = 0;
508:     obj->std->handlers = spl_handler_SplFixedArray;
509:     obj->std->ce = class_type;
510:     obj->std->flags = 0;
511:     obj->std->handlers = spl_handler_SplFixedArray;
512:     obj->std->ce = class_type;
513:     obj->std->flags = 0;
514:     obj->std->handlers = spl_handler_SplFixedArray;
515:     obj->std->ce = class_type;
516:     obj->std->flags = 0;
517:     obj->std->handlers = spl_handler_SplFixedArray;
518:     obj->std->ce = class_type;
519:     obj->std->flags = 0;
520:     obj->std->handlers = spl_handler_SplFixedArray;
521:     obj->std->ce = class_type;
522:     obj->std->flags = 0;
523:     obj->std->handlers = spl_handler_SplFixedArray;
524:     obj->std->ce = class_type;
525:     obj->std->flags = 0;
526:     obj->std->handlers = spl_handler_SplFixedArray;
527:     obj->std->ce = class_type;
528:     obj->std->flags = 0;
529:     obj->std->handlers = spl_handler_SplFixedArray;
530:     obj->std->ce = class_type;
531:     obj->std->flags = 0;
532:     obj->std->handlers = spl_handler_SplFixedArray;
533:     obj->std->ce = class_type;
534:     obj->std->flags = 0;
535:     obj->std->handlers = spl_handler_SplFixedArray;
536:     obj->std->ce = class_type;
537:     obj->std->flags = 0;
538:     obj->std->handlers = spl_handler_SplFixedArray;
539:     obj->std->ce = class_type;
540:     obj->std->flags = 0;
541:     obj->std->handlers = spl_handler_SplFixedArray;
542:     obj->std->ce = class_type;
543:     obj->std->flags = 0;
544:     obj->std->handlers = spl_handler_SplFixedArray;
545:     obj->std->ce = class_type;
546:     obj->std->flags = 0;
547:     obj->std->handlers = spl_handler_SplFixedArray;
548:     obj->std->ce = class_type;
549:     obj->std->flags = 0;
550:     obj->std->handlers = spl_handler_SplFixedArray;
551:     obj->std->ce = class_type;
552:     obj->std->flags = 0;
553:     obj->std->handlers = spl_handler_SplFixedArray;
554:     obj->std->ce = class_type;
555:     obj->std->flags = 0;
556:     obj->std->handlers = spl_handler_SplFixedArray;
557:     obj->std->ce = class_type;
558:     obj->std->flags = 0;
559:     obj->std->handlers = spl_handler_SplFixedArray;
560:     obj->std->ce = class_type;
561:     obj->std->flags = 0;
562:     obj->std->handlers = spl_handler_SplFixedArray;
563:     obj->std->ce = class_type;
564:     obj->std->flags = 0;
565:     obj->std->handlers = spl_handler_SplFixedArray;
566:     obj->std->ce = class_type;
567:     obj->std->flags = 0;
568:     obj->std->handlers = spl_handler_SplFixedArray;
569:     obj->std->ce = class_type;
570:     obj->std->flags = 0;
571:     obj->std->handlers = spl_handler_SplFixedArray;
572:     obj->std->ce = class_type;
573:     obj->std->flags = 0;
574:     obj->std->handlers = spl_handler_SplFixedArray;
575:     obj->std->ce = class_type;
576:     obj->std->flags = 0;
577:     obj->std->handlers = spl_handler_SplFixedArray;
578:     obj->std->ce = class_type;
579:     obj->std->flags = 0;
580:     obj->std->handlers = spl_handler_SplFixedArray;
581:     obj->std->ce = class_type;
582:     obj->std->flags = 0;
583:     obj->std->handlers = spl_handler_SplFixedArray;
584:     obj->std->ce = class_type;
585:     obj->std->flags = 0;
586:     obj->std->handlers = spl_handler_SplFixedArray;
587:     obj->std->ce = class_type;
588:     obj->std->flags = 0;
589:     obj->std->handlers = spl_handler_SplFixedArray;
590:     obj->std->ce = class_type;
591:     obj->std->flags = 0;
592:     obj->std->handlers = spl_handler_SplFixedArray;
593:     obj->std->ce = class_type;
594:     obj->std->flags = 0;
595:     obj->std->handlers = spl_handler_SplFixedArray;
596:     obj->std->ce = class_type;
597:     obj->std->flags = 0;
598:     obj->std->handlers = spl_handler_SplFixedArray;
599:     obj->std->ce = class_type;
600:     obj->std->flags = 0;
601:     obj->std->handlers = spl_handler_SplFixedArray;
602:     obj->std->ce = class_type;
603:     obj->std->flags = 0;
604:     obj->std->handlers = spl_handler_SplFixedArray;
605:     obj->std->ce = class_type;
606:     obj->std->flags = 0;
607:     obj->std->handlers = spl_handler_SplFixedArray;
608:     obj->std->ce = class_type;
609:     obj->std->flags = 0;
610:     obj->std->handlers = spl_handler_SplFixedArray;
611:     obj->std->ce = class_type;
612:     obj->std->flags = 0;
613:     obj->std->handlers = spl_handler_SplFixedArray;
614:     obj->std->ce = class_type;
615:     obj->std->flags = 0;
616:     obj->std->handlers = spl_handler_SplFixedArray;
617:     obj->std->ce = class_type;
618:     obj->std->flags = 0;
619:     obj->std->handlers = spl_handler_SplFixedArray;
620:     obj->std->ce = class_type;
621:     obj->std->flags = 0;
622:     obj->std->handlers = spl_handler_SplFixedArray;
623:     obj->std->ce = class_type;
624:     obj->std->flags = 0;
625:     obj->std->handlers = spl_handler_SplFixedArray;
626:     obj->std->ce = class_type;
627:     obj->std->flags = 0;
628:     obj->std->handlers = spl_handler_SplFixedArray;
629:     obj->std->ce = class_type;
630:     obj->std->flags = 0;
631:     obj->std->handlers = spl_handler_SplFixedArray;
632:     obj->std->ce = class_type;
633:     obj->std->flags = 0;
634:     obj->std->handlers = spl_handler_SplFixedArray;
635:     obj->std->ce = class_type;
636:     obj->std->flags = 0;
637:     obj->std->handlers = spl_handler_SplFixedArray;
638:     obj->std->ce = class_type;
639:     obj->std->flags = 0;
640:     obj->std->handlers = spl_handler_SplFixedArray;
641:     obj->std->ce = class_type;
642:     obj->std->flags = 0;
643:     obj->std->handlers = spl_handler_SplFixedArray;
644:     obj->std->ce = class_type;
645:     obj->std->flags = 0;
646:     obj->std->handlers = spl_handler_SplFixedArray;
647:     obj->std->ce = class_type;
648:     obj->std->flags = 0;
649:     obj->std->handlers = spl_handler_SplFixedArray;
650:     obj->std->ce = class_type;
651:     obj->std->flags = 0;
652:     obj->std->handlers = spl_handler_SplFixedArray;
653:     obj->std->ce = class_type;
654:     obj->std->flags = 0;
655:     obj->std->handlers = spl_handler_SplFixedArray;
656:     obj->std->ce = class_type;
657:     obj->std->flags = 0;
658:     obj->std->handlers = spl_handler_SplFixedArray;
659:     obj->std->ce = class_type;
660:     obj->std->flags = 0;
661:     obj->std->handlers = spl_handler_SplFixedArray;
662:     obj->std->ce = class_type;
663:     obj->std->flags = 0;
664:     obj->std->handlers = spl_handler_SplFixedArray;
665:     obj->std->ce = class_type;
666:     obj->std->flags = 0;
667:     obj->std->handlers = spl_handler_SplFixedArray;
668:     obj->std->ce = class_type;
669:     obj->std->flags = 0;
670:     obj->std->handlers = spl_handler_SplFixedArray;
671:     obj->std->ce = class_type;
672:     obj->std->flags = 0;
673:     obj->std->handlers = spl_handler_SplFixedArray;
674:     obj->std->ce = class_type;
675:     obj->std->flags = 0;
676:     obj->std->handlers = spl_handler_SplFixedArray;
677:     obj->std->ce = class_type;
678:     obj->std->flags = 0;
679:     obj->std->handlers = spl_handler_SplFixedArray;
680:     obj->std->ce = class_type;
681:     obj->std->flags = 0;
682:     obj->std->handlers = spl_handler_SplFixedArray;
683:     obj->std->ce = class_type;
684:     obj->std->flags = 0;
685:     obj->std->handlers = spl_handler_SplFixedArray;
686:     obj->std->ce = class_type;
687:     obj->std->flags = 0;
688:     obj->std->handlers = spl_handler_SplFixedArray;
689:     obj->std->ce = class_type;
690:     obj->std->flags = 0;
691:     obj->std->handlers
```



```

377:     return spl_fixedarray_object_read_dimension_helper(intern, offset);
378: }
379: /* }}} */
380:
381:
382: static inline void spl_fixedarray_object_write_dimension_helper(spl_fixedarray_object *intern, zval *offset, zval *value) /* {{{ */
383: {
384:     zend_long index;
385:
386:     if (!offset) {
387:         /* '[array[] = value]' syntax is not supported */
388:         zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
389:         return;
390:     }
391:
392:     if (Z_TYPE_P(offset) != IS_LONG) {
393:         index = spl_offset_convert_to_long(offset);
394:     } else {
395:         index = Z_LVAL_P(offset);
396:     }
397:
398:     if (index < 0 || index >= intern->array.size) {
399:         zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
400:         return;
401:     } else {
402:         if (!IS_UNDEF(intern->array.elements[index])) {
403:             zval_ptr_dtor(&(intern->array.elements[index]));
404:         }
405:         ZVAL_DEREF(value);
406:         ZVAL_COPY(&(intern->array.elements[index]), value);
407:     }
408: }
409: /* }}} */
410:
411: static void spl_fixedarray_object_write_dimension(zval *object, zval *offset, zval *value) /* {{{ */
412: {
413:     spl_fixedarray_object *intern;
414:     zval tmp;
415:
416:     intern = Z_SPLFIXEDARRAY_P(object);
417:
418:     if (intern->fptr_offset_set) {
419:         if (!offset) {
420:             ZVAL_NULL(&tmp);
421:             offset = &tmp;
422:         } else {
423:             SEPARATE_ARG_IF_REF(offset);
424:         }
425:         SEPARATE_ARG_IF_REF(value);
426:         zend_call_method_with_2_params(object, intern->std.ce, intern->fptr_offset_set, "offsetSet", NULL, offset, value);
427:         zval_ptr_dtor(value);
428:         zval_ptr_dtor(offset);
429:         return;
430:     }
431:
432:     spl_fixedarray_object_write_dimension_helper(intern, offset, value);
433: }
434: /* }}} */
435:
436: static inline void spl_fixedarray_object_unset_dimension_helper(spl_fixedarray_object *intern, zval *offset) /* {{{ */
437: {
438:     zend_long index;
439:
440:     if (Z_TYPE_P(offset) != IS_LONG) {
441:         index = spl_offset_convert_to_long(offset);
442:     } else {
443:         index = Z_LVAL_P(offset);
444:     }
445:
446:     if (index < 0 || index >= intern->array.size) {
447:         zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
448:         return;
449:     } else {
450:         zval_ptr_dtor(&(intern->array.elements[index]));
451:         ZVAL_UNDEF(&(intern->array.elements[index]));
452:     }
453: }
454: /* }}} */
455:
456: static void spl_fixedarray_object_unset_dimension(zval *object, zval *offset) /* {{{ */
457: {
458:     spl_fixedarray_object *intern;
459:
460:     intern = Z_SPLFIXEDARRAY_P(object);
461:
462:     if (intern->fptr_offset_del) {
463:         SEPARATE_ARG_IF_REF(offset);
464:         zend_call_method_with_2_params(object, intern->std.ce, intern->fptr_offset_del, "offsetUnset", NULL, offset);
465:         zval_ptr_dtor(offset);
466:         return;
467:     }
468:
469:     spl_fixedarray_object_unset_dimension_helper(intern, offset);
470: }
471: }
472: /* }}} */
473:
474: static inline int spl_fixedarray_object_has_dimension_helper(spl_fixedarray_object *intern, zval *offset, int check_empty) /* {{{ */
475: {
476:     zend_long index;
477:     int retval;
478:
479:     if (Z_TYPE_P(offset) != IS_LONG) {
480:         index = spl_offset_convert_to_long(offset);
481:     } else {
482:         index = Z_LVAL_P(offset);
483:     }
484:
485:     if (index < 0 || index >= intern->array.size) {
486:         retval = 0;
487:     } else {
488:         if (!IS_UNDEF(intern->array.elements[index])) {
489:             retval = 1;
490:         } else if (check_empty) {
491:             if (zend_is_true(intern->array.elements[index])) {
492:                 retval = 1;
493:             } else {
494:                 retval = 0;
495:             }
496:         } else /* != NULL and !check_empty */
497:             retval = 1;
498:     }
499: }
500:
501: return retval;
502: }
503: /* }}} */
504:
505: static int spl_fixedarray_object_has_dimension(zval *object, zval *offset, int check_empty) /* {{{ */
506: {
507:     spl_fixedarray_object *intern;
508:
509:     intern = Z_SPLFIXEDARRAY_P(object);
510:
511:     if (intern->fptr_offset_has) {
512:         zval rv;
513:         SEPARATE_ARG_IF_REF(offset);
514:         zend_call_method_with_2_params(object, intern->std.ce, intern->fptr_offset_has, "offsetExists", &rv, offset);
515:         zval_ptr_dtor(&rv);
516:         if (!IS_UNDEF(rv)) {
517:             zend_bool result = zend_is_true(rv);
518:             zval_ptr_dtor(&rv);
519:             return result;
520:         }
521:         return 0;
522:     }
523:
524:     return spl_fixedarray_object_has_dimension_helper(intern, offset, check_empty);
525: }
526: /* }}} */
527:
528: static int spl_fixedarray_object_count_elements(zval *object, zend_long *count) /* {{{ */
529: {
530:     spl_fixedarray_object *intern;
531:
532:     intern = Z_SPLFIXEDARRAY_P(object);
533:     if (intern->fptr_count) {
534:         zval rv;
535:         zend_call_method_with_0_params(object, intern->std.ce, intern->fptr_count, "count", &rv);
536:         if (!IS_UNDEF(rv)) {
537:             *count = zval_get_long(rv);
538:             zval_ptr_dtor(&rv);
539:         } else {
540:             *count = 0;
541:         }
542:     } else {
543:         *count = intern->array.size;
544:     }
545:     return SUCCESS;
546: }
547: /* }}} */
548:
549: /* {{{ proto void SplFixedArray::__construct([int size])
550: */
551: SPL_METHOD(SplFixedArray, __construct)
552: {
553:     zval *object = getThis();
554:     spl_fixedarray_object *intern;
555:     zend_long size = 0;
556:
557:     if (zend_parse_parameters_throw(ZEND_NUM_ARGS(), "|i", &size) == FAILURE) {
558:         return;
559:     }
560:
561:     if (size < 0) {
562:         zend_throw_exception_ex(spl_ce_InvalidArgumentException, 0, "array size cannot be less than zero");
563:         return;
564:     }

```

```

565:     intern = Z_SPLFIXEDARRAY_P(object);
566:
567:     if (intern->array.size > 0) {
568:         /* called __construct() twice, bail out */
569:         return;
570:     }
571: }
572:
573: spl_fixedarray_init(&intern->array, size);
574: }
575: /* }}} */
576:
577: /* {{{ proto void SplFixedArray::__wakeup()
578: */
579: SPL_METHOD(SplFixedArray, __wakeup)
580: {
581:     spl_fixedarray_object *intern = Z_SPLFIXEDARRAY_P(getThis());
582:     HashTable *intern_ht = zend_std_get_properties(getThis());
583:     zval *data;
584:
585:     if (zend_parse_parameters_none() == FAILURE) {
586:         return;
587:     }
588:
589:     if (intern->array.size == 0) {
590:         int index = 0;
591:         int size = zend_hash_num_elements(intern_ht);
592:
593:         spl_fixedarray_init(&intern->array, size);
594:
595:         ZEND_HASH_FOREACH_VAL(intern_ht, data) {
596:             ZVAL_COPY(&(intern->array.elements[index]), data);
597:             index++;
598:         }
599:         ZEND_HASH_FOREACH_END();
600:
601:         /* Remove the unserialized properties, since we now have the elements
602          * within the spl_fixedarray_object structure. */
603:         zend_hash_clean(intern_ht);
604:     }
605: }
606: /* }}} */
607:
608: /* {{{ proto int SplFixedArray::count(void)
609: */
610: SPL_METHOD(SplFixedArray, count)
611: {
612:     zval *object = getThis();
613:     spl_fixedarray_object *intern;
614:
615:     if (zend_parse_parameters_none() == FAILURE) {
616:         return;
617:     }
618:
619:     intern = Z_SPLFIXEDARRAY_P(object);
620:     return LONG(intern->array.size);
621: }
622: /* }}} */
623:
624: /* {{{ proto object SplFixedArray::toArray()
625: */
626: SPL_METHOD(SplFixedArray, toArray)
627: {
628:     spl_fixedarray_object *intern;
629:
630:     if (zend_parse_parameters_none() == FAILURE) {
631:         return;
632:     }
633:
634:     intern = Z_SPLFIXEDARRAY_P(getThis());
635:
636:     if (intern->array.size > 0) {
637:         int i = 0;
638:
639:         array_init(&return_value);
640:         for (; i < intern->array.size; i++) {
641:             if (!IS_UNDEF(intern->array.elements[i])) {
642:                 zend_hash_index_update(&ARRVAL_P(&return_value), i, &(intern->array.elements[i]));
643:             } else {
644:                 zend_hash_index_update(&ARRVAL_P(&return_value), i, &Z_UNINITIALIZED_VAL());
645:             }
646:         }
647:     } else {
648:         ZVAL_EMPTY_ARRAY(&return_value);
649:     }
650: }
651: /* }}} */
652:
653: /* {{{ proto object SplFixedArray::fromArray(array $data, bool $save_indexes)
654: */
655: SPL_METHOD(SplFixedArray, fromArray)
656: {
657:     zval *data;
658:     spl_fixedarray_array;
659:     spl_fixedarray_object *intern;
660:     int num;
661:     zend_bool save_indexes = 1;
662:
663:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "ab", &data, &save_indexes) == FAILURE) {
664:         return;
665:     }
666:
667:     num = zend_hash_num_elements(&ARRVAL_P(&data));
668:
669:     if (num > 0 & &save_indexes) {
670:         zval *element;
671:         zend_string *str_index;
672:         zend_long num_index, max_index = 0;
673:         zend_long tmp;
674:
675:         ZEND_HASH_FOREACH_KEY(&ARRVAL_P(&data), num_index, str_index) {
676:             if (str_index != NULL) { (zend_long)num_index < 0) {
677:                 zend_throw_exception_ex(spl_ce_InvalidArgumentException, 0, "array must contain only positive integer keys");
678:                 return;
679:             }
680:
681:             if (num_index > max_index) {
682:                 max_index = num_index;
683:             }
684:         }
685:         tmp = max_index + 1;
686:         if (tmp <= 0) {
687:             zend_throw_exception_ex(spl_ce_InvalidArgumentException, 0, "integer overflow detected");
688:             return;
689:         }
690:     }
691:
692:     spl_fixedarray_init(&array, tmp);
693:
694:     ZEND_HASH_FOREACH_KEY_VAL(&ARRVAL_P(&data), num_index, str_index, element) {
695:         ZVAL_DEREF(element);
696:         ZVAL_COPY(&(array.elements[num_index]), element);
697:     }
698:     ZEND_HASH_FOREACH_END();
699:
700:     if (num > 0 & &save_indexes) {
701:         zval *element;
702:         zend_long i = 0;
703:
704:         spl_fixedarray_init(&array, num);
705:
706:         ZEND_HASH_FOREACH_VAL(&ARRVAL_P(&data), element) {
707:             ZVAL_DEREF(element);
708:             ZVAL_COPY(&(array.elements[i]), element);
709:             i++;
710:         }
711:         ZEND_HASH_FOREACH_END();
712:     } else {
713:         spl_fixedarray_init(&array, 0);
714:     }
715:
716:     object_init_ex(&return_value, spl_ce_SplFixedArray);
717:
718:     intern = Z_SPLFIXEDARRAY_P(&return_value);
719:     intern->array = array;
720: }
721: /* }}} */
722:
723: /* {{{ proto int SplFixedArray::getSize(void)
724: */
725: SPL_METHOD(SplFixedArray, getSize)
726: {
727:     zval *object = getThis();
728:     spl_fixedarray_object *intern;
729:
730:     if (zend_parse_parameters_none() == FAILURE) {
731:         return;
732:     }
733:
734:     intern = Z_SPLFIXEDARRAY_P(object);
735:     return LONG(intern->array.size);
736: }
737: /* }}} */
738:
739: /* {{{ proto bool SplFixedArray::setSize(int size)
740: */
741: SPL_METHOD(SplFixedArray, setSize)
742: {
743:     zval *object = getThis();
744:     spl_fixedarray_object *intern;
745:     zend_long size;
746:
747:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "i", &size) == FAILURE) {
748:         return;
749:     }
750:
751:     if (size < 0) {
752:         zend_throw_exception_ex(spl_ce_InvalidArgumentException, 0, "array size cannot be less than zero");
753:         return;
754:     }

```

```

753: intern = _splfixedarray_p(object);
754:
755: spl_fixedarray_resize(intern->array, size);
756: RETURN_TRUE;
757: }
758: /* }}} */
759:
760: /* {{{ proto bool SplFixedArray::offsetExists(mixed $index)
761: Returns whether the requested $index exists. */
762: SPL_METHOD(SplFixedArray, offsetExists)
763: {
764:     zval *zindex;
765:     spl_fixedarray_object *intern;
766:
767:     IF_ZEND_PARSE_PARAMETERS(ZEND_NUM_ARGS(), "a", &zindex) == FAILURE {
768:         return;
769:     }
770:
771:     intern = _splfixedarray_p(getThis());
772:
773:     RETURN_BOOL(spl_fixedarray_object_has_dimension_helper(intern, zindex, 0));
774: } /* }}} */
775:
776: /* {{{ proto mixed SplFixedArray::offsetGet(mixed $index)
777: Returns the value at the specified $index. */
778: SPL_METHOD(SplFixedArray, offsetGet)
779: {
780:     zval *zindex, *value;
781:     spl_fixedarray_object *intern;
782:
783:     IF_ZEND_PARSE_PARAMETERS(ZEND_NUM_ARGS(), "a", &zindex) == FAILURE {
784:         return;
785:     }
786:
787:     intern = _splfixedarray_p(getThis());
788:     value = spl_fixedarray_object_read_dimension_helper(intern, zindex);
789:
790:     IF (value) {
791:         ZVAL_DEREF(value);
792:         ZVAL_COPY(return_value, value);
793:     } else {
794:         RETURN_NULL();
795:     }
796: } /* }}} */
797:
798: /* {{{ proto void SplFixedArray::offsetSet(mixed $index, mixed $newval)
799: Sets the value at the specified $index to $newval. */
800: SPL_METHOD(SplFixedArray, offsetSet)
801: {
802:     zval *zindex, *value;
803:     spl_fixedarray_object *intern;
804:
805:     IF_ZEND_PARSE_PARAMETERS(ZEND_NUM_ARGS(), "as", &zindex, &value) == FAILURE {
806:         return;
807:     }
808:
809:     intern = _splfixedarray_p(getThis());
810:     spl_fixedarray_object_write_dimension_helper(intern, zindex, value);
811:
812: } /* }}} */
813:
814: /* {{{ proto void SplFixedArray::offsetUnset(mixed $index)
815: Unsets the value at the specified $index. */
816: SPL_METHOD(SplFixedArray, offsetUnset)
817: {
818:     zval *zindex;
819:     spl_fixedarray_object *intern;
820:
821:     IF_ZEND_PARSE_PARAMETERS(ZEND_NUM_ARGS(), "a", &zindex) == FAILURE {
822:         return;
823:     }
824:
825:     intern = _splfixedarray_p(getThis());
826:     spl_fixedarray_object_unset_dimension_helper(intern, zindex);
827:
828: } /* }}} */
829:
830: static void spl_fixedarray_it_dtor(zend_object_iterator *iter) /* {{{ */
831: {
832:     spl_fixedarray_it *iterator = (spl_fixedarray_it *)iter;
833:
834:     zend_user_it_invalidate_current(iter);
835:     zval_ptr_dtor(&iterator->intern.it.data);
836: } /* }}} */
837:
838: static void spl_fixedarray_it_rewind(zend_object_iterator *iter) /* {{{ */
839: {
840:     spl_fixedarray_object *object = _splfixedarray_p(iterator->data);
841:
842:     IF (object->flags & SPL_FIXEDARRAY_OVERLOADED_REWIND) {
843:         zend_user_it_rewind(iter);
844:     } else {
845:         object->current = 0;
846:     }
847: } /* }}} */
848:
849: static int spl_fixedarray_it_valid(zend_object_iterator *iter) /* {{{ */
850: {
851:     spl_fixedarray_object *object = _splfixedarray_p(iterator->data);
852:
853:     IF (object->flags & SPL_FIXEDARRAY_OVERLOADED_VALID) {
854:         return zend_user_it_validate(iter);
855:     }
856:
857:     IF (object->current >= 0 && object->current < object->array.size) {
858:         return SUCCESS;
859:     }
860:     return FAILURE;
861: } /* }}} */
862:
863: static zval *spl_fixedarray_it_get_current_data(zend_object_iterator *iter) /* {{{ */
864: {
865:     zval *zindex;
866:     spl_fixedarray_object *object = _splfixedarray_p(iterator->data);
867:
868:     IF (object->flags & SPL_FIXEDARRAY_OVERLOADED_CURRENT) {
869:         return zend_user_it_get_current_data(iter);
870:     } else {
871:         zval *data;
872:
873:         ZVAL_LONG(&zindex, object->current);
874:
875:         data = spl_fixedarray_object_read_dimension_helper(object, &zindex);
876:         zval_ptr_dtor(&zindex);
877:
878:         IF (data == NULL) {
879:             data = &EG(uninitialized_zval);
880:         }
881:         return data;
882:     }
883: } /* }}} */
884:
885: static void spl_fixedarray_it_get_current_key(zend_object_iterator *iter, zval *key) /* {{{ */
886: {
887:     spl_fixedarray_object *object = _splfixedarray_p(iterator->data);
888:
889:     IF (object->flags & SPL_FIXEDARRAY_OVERLOADED_KEY) {
890:         zend_user_it_get_current_key(iter, key);
891:     } else {
892:         ZVAL_LONG(key, object->current);
893:     }
894: } /* }}} */
895:
896: static void spl_fixedarray_it_move_forward(zend_object_iterator *iter) /* {{{ */
897: {
898:     spl_fixedarray_object *object = _splfixedarray_p(iterator->data);
899:
900:     IF (object->flags & SPL_FIXEDARRAY_OVERLOADED_NEXT) {
901:         zend_user_it_move_forward(iter);
902:     } else {
903:         zend_user_it_invalidate_current(iter);
904:         object->current++;
905:     }
906: } /* }}} */
907:
908: /* {{{ proto int SplFixedArray::key()
909: Return current array key */
910: SPL_METHOD(SplFixedArray, key)
911: {
912:     spl_fixedarray_object *intern = _splfixedarray_p(getThis());
913:
914:     IF_ZEND_PARSE_PARAMETERS_NONE() == FAILURE {
915:         return;
916:     }
917:
918:     RETURN_LONG(intern->current);
919: } /* }}} */
920:
921: /* {{{ proto void SplFixedArray::next()
922: Move to next entry */
923: SPL_METHOD(SplFixedArray, next)
924: {
925:     spl_fixedarray_object *intern = _splfixedarray_p(getThis());
926:
927:     IF_ZEND_PARSE_PARAMETERS_NONE() == FAILURE {
928:         return;
929:     }
930:
931:     RETURN_LONG(intern->current++);
932: }

```

```

941: /* }}} */
942:
943: /* {{{ proto bool SplFixedArray::valid()
944: Check whether the datastructure contains more entries */
945: SPL_METHOD(SplFixedArray, valid)
946: {
947:     spl_fixedarray_object *intern = _splfixedarray_p(getThis());
948:
949:     IF_ZEND_PARSE_PARAMETERS_NONE() == FAILURE {
950:         return;
951:     }
952:
953:     RETURN_BOOL(intern->current >= 0 && intern->current < intern->array.size);
954: } /* }}} */
955:
956: /* {{{ proto void SplFixedArray::rewind()
957: Rewind the datastructure back to the start */
958: SPL_METHOD(SplFixedArray, rewind)
959: {
960:     spl_fixedarray_object *intern = _splfixedarray_p(getThis());
961:
962:     IF_ZEND_PARSE_PARAMETERS_NONE() == FAILURE {
963:         return;
964:     }
965:
966:     intern->current = 0;
967: } /* }}} */
968:
969: /* {{{ proto mixed SplFixedArray::current()
970: Return current datastructure entry */
971: SPL_METHOD(SplFixedArray, current)
972: {
973:     zval *zindex, *value;
974:     spl_fixedarray_object *intern = _splfixedarray_p(getThis());
975:
976:     IF_ZEND_PARSE_PARAMETERS_NONE() == FAILURE {
977:         return;
978:     }
979:
980:     ZVAL_LONG(&zindex, intern->current);
981:
982:     value = spl_fixedarray_object_read_dimension_helper(intern, &zindex);
983:
984:     zval_ptr_dtor(&zindex);
985:
986:     IF (value) {
987:         ZVAL_DEREF(value);
988:         ZVAL_COPY(return_value, value);
989:     } else {
990:         RETURN_NULL();
991:     }
992: } /* }}} */
993:
994: /* Iterator handler table */
995: static const zend_object_iterator_funcs spl_fixedarray_it_funcs = {
996:     spl_fixedarray_it_dtor,
997:     spl_fixedarray_it_valid,
998:     spl_fixedarray_it_get_current_data,
999:     spl_fixedarray_it_get_current_key,
1000:     spl_fixedarray_it_move_forward,
1001:     spl_fixedarray_it_rewind,
1002:     NULL
1003: };
1004:
1005: zend_object_iterator *spl_fixedarray_get_iterator(zend_class_entry *ce, zval *object, int by_ref) /* {{{ */
1006: {
1007:     spl_fixedarray_it *iterator;
1008:
1009:     IF (by_ref) {
1010:         zend_throw_exception(spl_ce_RuntimeException, "An iterator cannot be used with foreach by reference", 0);
1011:         return NULL;
1012:     }
1013:
1014:     iterator = emalloc(sizeof(spl_fixedarray_it));
1015:
1016:     zend_iterator_init(&zend_object_iterator*(iterator));
1017:
1018:     ZVAL_COPY(&iterator->intern.it.data, object);
1019:     iterator->intern.it.funcs = &spl_fixedarray_it_funcs;
1020:     iterator->intern.ce = ce;
1021:     ZVAL_UNDEF(&iterator->intern.value);
1022:
1023:     return iterator->intern.it;
1024: } /* }}} */
1025:
1026: /* {{{ */
1027:
1028: /* {{{ */
1029:
1030: ZEND_ARG_INFO_EX(arginfo_splfixedarray_construct, 0, 0, 0)
1031: ZEND_ARG_INFO(0, size)
1032: ZEND_ARG_INFO(0, offset)
1033:
1034: ZEND_ARG_INFO_EX(arginfo_splfixedarray_offsetGet, 0, 0, 1)
1035: ZEND_ARG_INFO(0, index)
1036: ZEND_ARG_INFO(0, index)
1037:
1038: ZEND_ARG_INFO_EX(arginfo_splfixedarray_offsetSet, 0, 0, 2)
1039: ZEND_ARG_INFO(0, index)
1040: ZEND_ARG_INFO(0, newval)
1041: ZEND_ARG_INFO(0, newval)
1042:
1043: ZEND_ARG_INFO_EX(arginfo_splfixedarray_setSize, 0)
1044: ZEND_ARG_INFO(0, value)
1045: ZEND_ARG_INFO(0, value)
1046:
1047: ZEND_ARG_INFO_EX(arginfo_splfixedarray_fromArray, 0, 0, 1)
1048: ZEND_ARG_INFO(0, data)
1049: ZEND_ARG_INFO(0, save_indexes)
1050: ZEND_ARG_INFO(0, save_indexes)
1051:
1052: ZEND_ARG_INFO_EX(arginfo_splfixedarray_void, 0)
1053: ZEND_ARG_INFO(0, )
1054:
1055: static const zend_function_entry spl_funcs_splfixedarray[] = { /* {{{ */
1056:     SPL_ME(splfixedarray, __construct, arginfo_splfixedarray_construct, ZEND_ACC_PUBLIC)
1057:     SPL_ME(splfixedarray, __wakeup, arginfo_splfixedarray_void, ZEND_ACC_PUBLIC)
1058:     SPL_ME(splfixedarray, count, arginfo_splfixedarray_void, ZEND_ACC_PUBLIC)
1059:     SPL_ME(splfixedarray, toArray, arginfo_splfixedarray_void, ZEND_ACC_PUBLIC)
1060:     SPL_ME(splfixedarray, fromArray, arginfo_splfixedarray_fromArray, ZEND_ACC_PUBLIC | ZEND_ACC_STATIC)
1061:     SPL_ME(splfixedarray, setSize, arginfo_splfixedarray_setSize, ZEND_ACC_PUBLIC)
1062:     SPL_ME(splfixedarray, setSize, arginfo_splfixedarray_setSize, ZEND_ACC_PUBLIC)
1063:     SPL_ME(splfixedarray, offsetExists, arginfo_splfixedarray_offsetGet, ZEND_ACC_PUBLIC)
1064:     SPL_ME(splfixedarray, offsetGet, arginfo_splfixedarray_offsetGet, ZEND_ACC_PUBLIC)
1065:     SPL_ME(splfixedarray, offsetSet, arginfo_splfixedarray_offsetSet, ZEND_ACC_PUBLIC)
1066:     SPL_ME(splfixedarray, offsetUnset, arginfo_splfixedarray_offsetSet, ZEND_ACC_PUBLIC)
1067:     SPL_ME(splfixedarray, rewind, arginfo_splfixedarray_void, ZEND_ACC_PUBLIC)
1068:     SPL_ME(splfixedarray, key, arginfo_splfixedarray_void, ZEND_ACC_PUBLIC)
1069:     SPL_ME(splfixedarray, next, arginfo_splfixedarray_void, ZEND_ACC_PUBLIC)
1070:     SPL_ME(splfixedarray, valid, arginfo_splfixedarray_void, ZEND_ACC_PUBLIC)
1071:     PHP_FE_END
1072: };
1073:
1074: /* {{{ PHP_MINIT_FUNCTION */
1075:
1076: /* {{{ PHP_MINIT_FUNCTION */
1077: PHP_MINIT_FUNCTION(spl_fixedarray)
1078: {
1079:     REGISTER_SPL_STM_CLASS_EX(splFixedArray, spl_fixedarray_new, spl_funcs_splfixedarray);
1080:     memory(spl_handler_splfixedarray, zend_get_std_object_handlers(), sizeof(zend_object_handlers));
1081:
1082:     spl_handler_splfixedarray.offset = &offsetOf(spl_fixedarray_object, scd);
1083:     spl_handler_splfixedarray.clone_obj = &spl_fixedarray_object_clone;
1084:     spl_handler_splfixedarray.read_dimension = &spl_fixedarray_object_read_dimension;
1085:     spl_handler_splfixedarray.write_dimension = &spl_fixedarray_object_write_dimension;
1086:     spl_handler_splfixedarray.unset_dimension = &spl_fixedarray_object_unset_dimension;
1087:     spl_handler_splfixedarray.has_dimension = &spl_fixedarray_object_has_dimension;
1088:     spl_handler_splfixedarray.count_elements = &spl_fixedarray_object_count_elements;
1089:     spl_handler_splfixedarray.get_properties = &spl_fixedarray_object_get_properties;
1090:     spl_handler_splfixedarray.get_gc = &spl_fixedarray_object_get_gc;
1091:     spl_handler_splfixedarray.dtor_obj = &zend_object_std_dtor_obj;
1092:     spl_handler_splfixedarray.free_obj = &spl_fixedarray_object_free_storage;
1093:
1094:     REGISTER_SPL_IMPLEMENT(splFixedArray, Iterator);
1095:     REGISTER_SPL_IMPLEMENT(splFixedArray, ArrayAccess);
1096:     REGISTER_SPL_IMPLEMENT(splFixedArray, Countable);
1097:
1098:     spl_ce_splfixedarray->get_iterator = &spl_fixedarray_get_iterator;
1099:
1100:     return SUCCESS;
1101: } /* }}} */
1102:
1103: /* {{{ */
1104:
1105: /* Local variables:
1106: * tab-width: 4
1107: * c-basic-offset: 4
1108: * End
1109: * vim600: noet sw=4 ts=4 fdm=marker
1110: * vim600: noet sw=4 ts=4
1111: */

```

```
1: /*
2:  *-----
3:  * | PHP Version 7 |
4:  *-----
5:  * | Copyright (c) 1997-2018 The PHP Group |
6:  *-----
7:  * | This source file is subject to version 3.01 of the PHP license, |
8:  * | that is bundled with this package in the file LICENSE, and is |
9:  * | available through the world-wide-web at the following uri: |
10:  * | http://www.php.net/license/3_01.txt |
11:  * | If you did not receive a copy of the PHP license and are unable to |
12:  * | obtain it through the world-wide-web, please send a note to |
13:  * | license@php.net so we can mail you a copy immediately. |
14:  *-----
15:  * | Author: Antony Duvall <trond@daylessday.org> |
16:  * | Elienne Kneuss <colder@php.net> |
17:  *-----
18:  */
19:
20: /* $Id$ */
21:
22: #ifndef SPL_FIXEDARRAY_H
23: #define SPL_FIXEDARRAY_H
24:
25: extern PHPAPI zend_class_entry *spl_ce_SplFixedArray;
26:
27: PHP_MINIT_FUNCTION(spl_fixedarray);
28:
29: #endif /* SPL_FIXEDARRAY_H */
30:
31: /*
32:  * Local Variables:
33:  * tab-width: 4
34:  * c-basic-offset: 4
35:  * End:
36:  * vim600: noet sw=4 ts=4 fM=marker
37:  * vim600: noet sw=4 ts=4
38:  */
```

```
1: /*
2:  * -----
3:  * | PHP Version 7 |
4:  * -----
5:  * | Copyright (c) 1997-2018 The PHP Group |
6:  * -----
7:  * | This source file is subject to version 3.01 of the PHP license, |
8:  * | that is bundled with this package in the file LICENSE, and is |
9:  * | available through the world-wide-web at the following url: |
10:  * | http://www.php.net/license/3.01.txt |
11:  * | If you did not receive a copy of the PHP license and are unable to |
12:  * | obtain it through the world-wide-web, please send a note to |
13:  * | license@php.net so we can mail you a copy immediately. |
14:  * -----
15:  * | Authors: Marcus Boerger <helly@php.net> |
16:  * -----
17:  */
18:
19: #ifndef PHP_SPL_H
20: #define PHP_SPL_H
21:
22: #include "php.h"
23: #include "stdarg.h"
24:
25: #define PHP_SPL_VERSION PHP_VERSION
26:
27: #extern zend_module_entry spl_module_entry;
28: #define phpext_spl_ptr spl_module_entry
29:
30: #ifndef PHP_WIN32
31: # if defined SPL_EXPORTS
32: #  define SPL_API __declspec(dllexport)
33: # elif defined(COMPILE_DL_SPL)
34: #  define SPL_API __declspec(dllimport)
35: # else
36: #  define SPL_API /* nothing */
37: # endif
38: #elif defined(__GNUC__) && __GNUC__ >= 4
39: # define SPL_API __attribute__((visibility("default")))
40: #else
41: # define SPL_API
42: #endif
43:
44: #if defined(PHP_WIN32) && !defined(COMPILE_DL_SPL)
45: #undef phpext_spl
46: #define phpext_spl NULL
47: #endif
48:
49: #PHP_MINIT_FUNCTION(spl);
50: #PHP_MSHUTDOWN_FUNCTION(spl);
51: #PHP_RINIT_FUNCTION(spl);
52: #PHP_ASHUTDOWN_FUNCTION(spl);
53: #PHP_MINFO_FUNCTION(spl);
54:
55:
56: #END_BEGIN_MODULE_GLOBALS(spl)
57: zend_string *autoload_extensions;
58: HashTable *autoload_functions;
59: intptr_t hash_mask_handle;
60: intptr_t hash_mask_handlers;
61: int hash_mask_init;
62: int autoload_running;
63: #END_END_MODULE_GLOBALS(spl)
64:
65: #END_EXTERN_MODULE_GLOBALS(spl)
66: #define SPL_G(v) ZEND_MODULE_GLOBALS_ACCESSOR(spl, v)
67:
68: #PHP_FUNCTION(spl_classes);
69: #PHP_FUNCTION(class_parents);
70: #PHP_FUNCTION(class_implements);
71: #PHP_FUNCTION(class_used);
72:
73: #PHPAPI zend_string *php_spl_object_hash(zval *obj);
74:
75: #endif /* PHP_SPL_H */
76:
77: /*
78:  * Local Variables:
79:  * -tab-width: 4
80:  * -tab-width: 4
81:  * End:
82:  * vim: set fdm=marker
83:  * vim: noet sw=4 ts=4
84:  */
```